Muistin siivous

Zoltán Varga

Tampereen yliopisto
Tietojenkäsittelytieteiden laitos
Tietojenkäsittelyoppi /
Ohjelmistokehitys
Pro gradu -tutkielma
Huhtikuu 2006

Tampereen yliopisto
Tietojenkäsittelytieteiden laitos
Tietojenkäsittelyoppi / Ohjelmistokehitys
Tekijän Nimi: Zoltán Varga
Pro gradu -tutkielma, 64 sivua
Huhtikuu 2006

Tutkielmassa esitellään roskan käsite tietojenkäsittelytieteessä, roskienkeruun keskeiset käsitteet ja perusmenetelmät muunnelmineen sekä nykyaikaiset tehokkaat algoritmit. Keskipisteenä ovat kuitenkin muistinhallintatutkimuksen 2000-luvun saavutukset, tutkimusaiheet ja tutkimusvälineet. Näitä hyödyntää tutkielmassa esiteltävä uusi CBRC-roskienkeruualgoritmi. Lisäksi katsastetaan ohjelmoijan vastuu automaattisessa muistinhallinnassa sekä ohjelmoinnissa käytettävissä olevat roskienkeruutietoiset välineet eräissä ohjelmointikielissä ja –ympäristöissä (Java, .Net, C++).

Avainsanat ja -sanonnat: roskienkeruu, muistinsiivous, muistinhallinta, algoritmit, ohjelmointikielet
CR-luokat: D 3.4, D.4.2, D.3.3

Sisällys

1. Johdanto

Roskienkeruun (*garbage collection*[1]) tarkoitus on löytää ja poistaa tarpeettomat alkiot tutkittavasta joukosta. Sen tunnetuin sovellus on ohjelmien automaattisen muistinhallinnan toteutus, eli muistinsiivous, joka on tämän katsauksen teemana. Ensimmäiset ohjelmointikielen ajoympäristöön sulautetut alijärjestelmät, joiden tehtävänä oli päätellä mitkä ohjelman varaamat muistisolut olivat tarpeellisia ja mitkä eivät, kehitettiin jo 1950-luvun lopussa LISP-ohjelmointikieltä varten. Alkuperäiset toteutukset olivat varsin tehottomia, mutta automaattisen muistinhallinnan edut manuaaliseen nähden kannustivat menetelmien kehittämiseen. Nykyään roskienkeruu on valmiiksi integroitu ohjelmistotuotannon valtavirran kieliin ja alustoihin (Java, .Net), tai ainakin se on sovellettavissa niihin ajoaikaisen kirjaston tai muun vastaavan mekanismin avulla (C/C++).

1.1. Roskienkeruun tarve ja merkitys

Tietokoneohjelmilla on tavallisesti käytössään kaksi muistiavaruutta: pino (*stack*) käännösaikana tunnettuja paikallismuuttujia varten ja keko (*heap*) ajoaikana dynaamisesti varattua muistia varten. Pinon hallinta on melko suoraviivaista, mutta keon muistinhallinta voidaan ratkaista monella tavalla. Muistinhallinta on kuitenkin joko eksplisiittistä (manuaalista) tai implisiittistä (automaattista).

Vanhanmalliset imperatiiviset kielet jättävät ohjelmoijan vastuulle dynaamisesti varatun muistin vapauttamisen. Näissä kielissä puuttuva muistin vapautusoperaatio aiheuttaa muistivuodon (*memory leak*), kun taas muistialueen ennenaikainen vapautus synnyttää roikkuvia osoittimia (*dangling pointer*), jotka voivat aiheuttaa epädeterminististä ohjelman käyttäytymistä. Manuaalinen muistinhallinta lienee ohjelmavirheiden merkittävin lähde.

Modulaarisessa ohjelmoinnissa eksplisiittinen muistinhallinta voi synnyttää ohjelmalogiikasta irrallisia riippuvuussuhteita moduulien väliin. Varatun muistin hallinta ja saatavuustietojen ylläpito voivat muuttua ohjelmassa paikallisesta ongelmasta globaaliseksi, mikä vähentää uudelleenkäytön mahdollisuuksia sekä heikentää moduulien ylläpidettävyyttä ja laajennettavuutta. Huonona vaihtoehtona riippuvuuksille on luoda datasta oma kopio jokaiselle moduulille. Ohjelmoija saattaa törmätä myös sellaiseen ongelmaan, että ohjelman rakenteesta johtuen ei löydy kaikilta osin sopivaa paikkaa tietyn muistialkion vapauttamiseen.

---

[1] Tutkielmassa alkuperäinen englanninkielinen termi esitellään suomenkielisen termin tai kuvauksen perässä sulkeissa ja kursivoituna. Osa suomennoksista ovat epävirallisia.

Eksplisiittinen muistinhallinta ilmenee siis ongelmana monessa ohjelmistokehityksen vaiheessa. Ratkaisua näihin ongelmiin on haettu automaattisesta muistinhallinnasta. Roskienkeruu hoitaa tarpeettomaksi käyneen varatun muistin vapauttamisen, mutta se ei kuitenkaan kokonaan nosta vastuuta muistinhallinnasta ohjelmoijan harteilta.

Moni sovellus hallinnoi muistin ulkopuolisia voimavaroja, mikä vaatii vuorovaikutusta sovelluksen ja roskienkeruumekanismin välillä. Sovelluksen ja suoritusympäristön välinen vuorovaikutus on abstrahoitu eri ohjelmointikielissä eri tavoilla ja tasoilla. Roskienkeruuseen liittyvät prosessit tapahtuvat yleensä sekä automaattisesti että implisiittisesti, ja tapahtumien kulkuun voidaan vaikuttaa kielestä riippumatta vain melko suppean välineistön avulla. Siksi suoritusjärjestelmän (*run-time system*) roskienkeruumekanismin toimintaperiaatteet ja olettamukset on syytä tuntea tarkkaan ja ottaa huomioon ohjelmoinnissa. Ei pidä myöskään unohtaa, että ohjelmoijan on edelleen osoitettava, mitkä muistialkiot ovat ohjelmassa tarpeettomia, jotta roskienkeruu toimisi. Ohjelmointikielten erilaisten välineiden avulla voidaan edesauttaa muistin tehokasta kierrätystä.

Ohjelmia tuotetaan hyvin monenlaisiin tarkoituksiin, siksi niille asetetut vaatimukset läpäisykyvyn ja muistinkäytön suhteen vaihtelevat suuresti. Koska roskienkeruusta voi muodostua merkittävä pullonkaula ohjelman suorituksessa, on syytä ottaa käyttöön eri suoritusympäristöjen tarjoamat mahdollisuudet virittää ja tehostaa roskienkeruu. Esimerkiksi Java-alustalla voidaan valita sopiva roskienkerääjä monesta vaihtoehdosta, mutta muissakin ympäristöissä on runsaasti optimoitavia tekijöitä. Roskienkeruumenetelmien toimintamekanismin tarkka tuntemus on tarpeen tehtäessä roskienkeruun hienosäätöä. Menetelmien vertailu taas edesauttaa kunkin menetelmän hyötyjen ja haittapuolten ymmärtämistä, mikä on välttämätöntä oikean menetelmän valinnassa.

1.2. Tutkielman rakenne

Tutkielma jakaantuu sisällöltään kolmeen osaan. Ensimmäisissä luvuissa tutustutaan ensin roskienkeruun aihepiiriin yleisesti katsastaen klassisia algoritmeja ja teorioita tarkoituksena esitellä kaikkia tunnettuja perusmenetelmiä. Toisessa osassa siirrytään tuoreimpiin tutkimustuloksiin ja hahmotellaan oman roskienkeruumenetelmän toimintaperiaatteita. Lopuksi selvitellään eri ohjelmointikielten ja -alustojen suhdetta automaattiseen muistinhallintaan.

Ensimmäiseen osaan kuuluvat luvut 2 - 6. Luvussa 2 esitellään roskan käsite, automaattisen muistinhallinnan perusolettamukset ja hypoteesit sekä roskienkeruumenetelmien pieni taksonomia. Luvusta 3 saa taustatietoja

muistinvaraustekniikoista, optimoinnista ja muistinhallinnan ongelmista, jotka valaisevat luvussa 4 esiteltäviä kanonisia menetelmiä ja niiden muunnelmien kirjoa. Kanoniset menetelmät ovat kehittyneempien tekniikoiden, kuten luvun 5 vähittäisten ja luvun 6 ikäpohjaisten menetelmien, perustana.

Roskienkeruututkimusta ovat edistäneet suuresti 2000-luvulla yhtenäistyneistä työvälineistä saatavat vertailukelpoiset tulokset. Toisen osan luvussa 7 tutustutaan roskienkeruututkimuksessa käytettäviin nykyaikaisiin työvälineisiin, Jikes RVM -virtuaalikoneeseen ja suosittuihin SPEC benchmark -testeihin. Merkittävimpiin 2000-luvun tutkimustuloksiin tutustutaan luvuissa 8 ja 9. Luvussa 8 tutustutaan roskienkeruututkimuksen uuteen, yhdistävyysanalyysiin perustuvaan haaran, joka sai alkunsa ikäpohjaisia menetelmiä ennustettavamman suorituksen tavoittelusta. Toistaiseksi yhdistävyyspohjainen roskienkeruu ja luvun 9 menetelmät ovat ainoastaan akateemisia ilmiöitä. Luku 9 sisältää useita lyhyehköjä kuvauksia uusista mielenkiintoisista tuloksista ja menetelmistä. Tutkielmassa hahmotellaan myös uutta tekniikkaa, jossa on sovellettu yhdistävyyspohjaisen roskienkeruun tuomat mahdollisuudet tunnettuun viittauslaskenta-algoritmiin. Menetelmää esitellään luvussa 10.

Tutkielman loppuosassa keskitytään automaattisen muistinhallinnan ohjelmointiteknisiin näkökohtiin. Asiaa tarkastellaan luvussa 11 yleisellä tasolla, seuraavissa luvuissa yksittäisten ohjelmointikielten ja -alustojen kautta: luku 12 tutustuttaa .Net-maailman asioihin, luku 13 roskienkeruuseen Java-alustalla ja lopuksi luku 14 erääseen C++-kielen roskienkeruulaajennokseen. Luku 15 on tutkielman yhteenveto.

Tarkoituksenani on esitellä roskienkeruu yleisellä tasolla, tavallisessa ohjelmaympäristössä, siksi erikoisympäristöt, kuten hajautetut ja reaaliaikaiset ympäristöt jäävät tämän tutkielman aihepiirin ulkopuolelle. Roskienkeruuta hajautetuissa ympäristöissä koskevat täsmälleen samat ongelmat kuin muutakin tietojenkäsittelyä. Reaaliaikaisuus on roskienkeruussa erityinen haaste taattujen vasteaikojen takia, mutta tutkielman tarkoitus on esittää kattavasti roskienkeruun kaikki muutkin haasteet ja ongelmat.

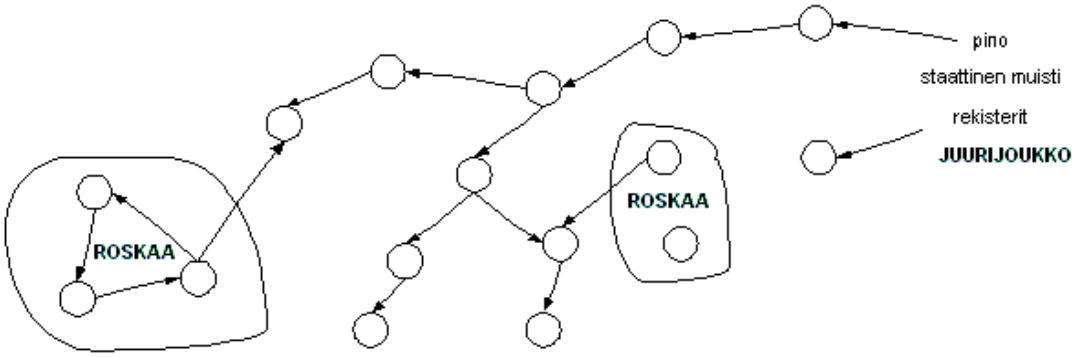
2. Roskienkeruun keskeiset käsitteet, menetelmät ja hypoteesit

Ennen kuin tutustutaan algoritimien kirjoon, on syytä esitellä muutama roskienkeruun aihepiiriin kuuluva käsite ja pieni algoritmien taksonomia.

2.1. Roskan käsite

Muistinsiivous perustuu seuraavaan periaatteeseen: jos johonkin muistialkioon ei pääse ohjelmasta osoittamaan mitään viittauspolkua pitkin, alkio voidaan

turvallisesti merkitä roskaksi (*garbage*). Kuten kuva 1 havainnollistaa, roskienkerääjä eli siivoin (*collector*) pitää tarpeellisina ohjelman ulottuvissa olevia alkioita (*reachable data*) ja tarpeettomina (roskina) kaikkia irrallisia alkioita (*unreachable data*). Alkiolla tarkoitetaan yleisesti ohjelman dynaamisesti varaamaa muistialuetta, esimerkiksi oliopohjaisten kielten oliota. Tutkielmassa viittaan kaikkiin muistialkioihin olioina. Olio on siis muistialkion abstraktio. Oliolla on tila, joka on tallennettu tietylle muistialueelle.



Kuva 1 Graafin solmut, joihin ei pääse juurijoukosta, ovat roskia.

Roskienkerääjän kaksi tehtävää ovat roskien löytäminen (*discovery*) ja siivoaminen (lakaisu), eli roskaksi todettujen olioiden varaaman muistin vapauttaminen (*reclamation*). Algoritmista riippuen tehtävät voidaan suorittaa peräkkäin tai lomittain, tai siivoaminen voidaan sivuuttaa väliaikaisesti otollisempaa hetkeä odottaen (*deferred reclamation*). Siivoaminen voi tapahtua myös implisiittisesti ilman erillisiä toimenpiteitä, kuten tullaan algoritmien esittelyn yhteydessä näkemään.

2.2. Kanoniset roskienkeruumenetelmät

Roskien löytämiseen on tarjolla kaksi perustekniikkaa. Toinen niistä on viittauslaskureiden käyttö (*reference counting collection*) ja toinen olioverkon läpikäynti (*tracing collection*), jonka kaksi perusalgoritmia ovat kaksivaiheinen "merkitse ja lakaise" (*mark-and-sweep*) ja “pysäytä ja kopioi” (*stop-and-copy*).

Viittauslaskurialgoritmi toimii laskennan lomassa. Jokaiselle oliolle ylläpidetään laskuria, jota kasvatetaan yhdellä aina, kun olioon luodaan uusi viittaus, ja vastaavasti vähennetään yhdellä, kun viittaus poistetaan. Viitelaskuritekniikan roskamäärittely on yksinkertainen: olio on roskaa, jos sen viitelaskurin arvo on nolla.

Viittauslaskennan etu graafin läpikäyntiin nähden on sen vähittäisyys. Hyötylaskenta ja roskienkeruu lomittuvat synkronisesti. Jälkimmäinen tekniikka perusmuodossaan taas pysäyttää hyötylaskennan kunnes

roskienkeruu on suoritettu alusta loppuun, eli se toimii niin sanotulla *stop-and-go* tai *stop-the-world* –periaatteella. Tekniikasta on kuitenkin olemassa myös vähittäisiä (*incremental*) ja samanaikaisia (*concurrent*) muunnelmia. Samanaikainen (rinnakkainen) roskienkerääjä toimii ohjelman hyötylaskentaa tekevän osan, eli muuttimen (*mutator* [Dijkstra *et al.*, 1978]) kanssa asynkronisesti, vähittäisen roskienkeruun ja muuttimen toiminta lomittuvat synkronisesti.

Olioverkon läpikäyntiä käyttävä tekniikka perustuu graafiteoriaan. Ohjelman käyttämä muisti voidaan nähdä suunnattuna verkkona (*GOG, Global Object Graph*), jonka solmuina ovat ohjelman luomat oliot, ja kaarina niitä yhdistävät viittaukset. Verkko käydään läpi lähtien viittausjuurijoukosta (*root set*), jonka muodostavat staattisella muistialueella, kontrollipinossa ja rekistereissä olevat viittausmuuttujat, olioiden jäsenmuuttujat mukaanlukien. Tarpeellisuutta laajasti määritellen voidaan sanoa, että käytyään läpi kaikki juurijoukosta lähtevät viittauspolut roskienkerääjä tuntee tarpeellisten olioiden joukon. Nämä polut muodostavat siis transitiivisen sulkeuman (*transitive closure*). Oliot, jotka jäävät sulkeuman ulkopuolelle, ovat roskia. Tämä tarpeellisuuden määritelmä on sikäli laaja, että vaikka ohjelmalla olisikin periaatteessa mahdollisuus käyttää tiettyä oliota, sellaista suorituspolkua, jossa oliota käytettäisiin, ei välttämättä enää ole. Edellä mainittu joukko saattaa siis sisältää myös kuollutta dataa (*dead data*), jota sanotaan tässä yhteydessä semanttiseksi roskaksi (*semantic garbage*). Roskienkeruussa datan elollisuuden (*liveness*) kriteeri on varsin konservatiivinen ja staattinen esimerkiksi optimoivan JIT-kääntäjän kriteereihin nähden.

Olioverkon läpikäynti vaatii sen, että pystytään tunnistamaan ajoaikana joko keko-olioiden tyyppi tai ainakin osoittimet tietueissa ja paikallismuuttujien joukossa. Edellinen vaatii yleensä kääntäjän tukea. Olioihin liitetään ohjelmalle näkymätön tyyppikenttä, josta olion formaatti voidaan päätellä tarkasti. Tarkkojen roskienkerääjien (*precise garbage collector*) lisäksi on olemassa myös konservatiivisia kerääjiä (*conservative garbage collector*), jotka eivät tiedä muistin tarkkaa sisältöä (koska kieli ei oletusarvoisesti tue roskienkeruuta), vaan olettavat, että mikä tahansa kenttä, joka ”näyttää” osoittimelta, on myös sellainen. Kaikki tutkielmassa esitetyt menetelmät perustuvat tyyppitietoisuuteen (*type-safety*), vasta luvussa 14 tutustutaan konservatiiviseen kerääjään.

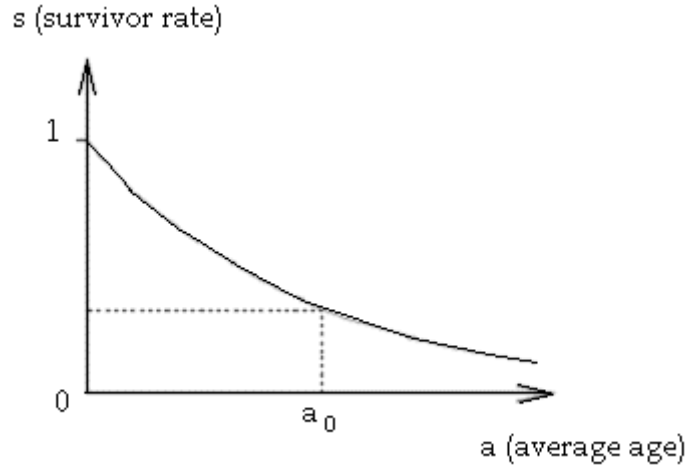
Perinteisessä stop-and-go -mallissa järjestelmän päätettäväksi jää roskienkeruun tahdistus. Yleensä toimenpide suoritetaan vasta silloin, kun muisti loppuu (*heap exhaustion*), eli muistinvarauspyyntö epäonnistuu. Vaihtoehtona tällaiselle *fixed-space* -menetelmälle on prosentteina ilmoitettu muistin-

käyttökynnys (*allocation-threshold*), jota ohjelma ei saa ylittää. Moni algoritmi toimii tehokkaasti vain, jos sillä on tarpeeksi vapaata työmuistia käytettävissään. Varhaisten kerääjien tärkein säädettävä ominaisuus olikin yllämainittu allokaatiokynnys, joka oli samalla merkittävin tapa optimoida roskienkeruu. Kehittyneemmät algoritmit ovat paljon laajemmin optimoitavissa.

2.3. Heikko sukupolvihypoteesi

Kirjallisuudessa mainitaan usein havainto olioiden lyhytikäisyydestä, eli olioiden suuresta lapsikuolleisuudesta (*infant mortality*), jonka on alunperin tehnyt Ungar [1984] heikossa sukupolvihypoteesissään (*weak generational hypothesis)*. Nykyaikaisten tehomenetelmien perustana oleva ikäpohjainen roskienkeruu (*generational garbage collection*), jota käsitellään tarkemmin luvussa 6, on saanut alkunsa tästä lyhytikäisyyshavainnosta.

Intuitio "mitä pitempään ohjelman suoritus jatkuu, sitä enemmän olioita kuolee" pitää yleensä hyvin paikkaansa, mutta Stefanović ja kumppanit [2000] osoittivat, että täysin tyydyttävää analyyttistä mallia olioiden ikäjakaumasta ei ole olemassa. Siitä huolimatta paljon käytetty ohjelman olioiden ikäjakauma-malli on niin sanottu radioaktiivinen hajoamismalli (*radioactive decay model*), joka kuvaa ikähajontaa olioiden keskivertoiän käänteisenä eksponentiaalisena funktiona, jonka kuvaaja on esitettynä kuvassa 2.



Kuva 2 Radioaktiivinen hajoamismalli: s-kordinaatti ilmaisee olioiden selviämistodennäköisyyden, a-kordinaatti olioiden keskivertoikää. [Hirzel *et al.*, 2003]

Myöhemmissä luvuissa tullaan näkemään, kuinka keskeinen merkitys on olioiden demografialla (keskivertoiällä ja ikäjakaumalla) roskienkeruun opportunistisen tahdistuksen ja kohdistuksen kannalta.

3. Roskienkeruun haasteet ja mittarit

Roskienkeruun suhteen voidaan (hieman yksinkertaistettuna) esittää kolmenlaista tehokkuusnäkökohtaa: aikakustannukset, tilakustannukset ja katkojen pituus hyötylaskennassa. Roskienkeruualgoritmien laaja kirjo selittyykin sillä, että mikään algoritmi ei voi olla samaan aikaan kaikkien näkökohtien kannalta erityisen tehokas. Tämän luvun alussa puhutaan roskienkeruun optimoinnista ja erilaisista tavoista mitata ja vertailla tehokkuuksia. Koska muistinvaraus- ja roskienkeruumekanismi kulkevat useimmiten käsi kädessä, katsastetaan myös erilaisia muistinvarausmekanismeja ja niiden vaikutusta roskienkeruun tehokkuuteen. Tässä luvussa myös esitellään, millaisia positiivisia (muistinkäytön tehokkuutta edistäviä) vaikutuksia roskienkeruulla voi olla.

3.1. Optimointi

Algoritmin optimointi tehdään aina jonkin ominaisuuden suhteen toisen ominaisuuden kustannuksella. Optimoitava tekijä on yleensä joko ohjelman läpäisykyky (*throughput*) tai sen muistinkäyttö. Roskienkeruussa keskeisiksi tekijöiksi nousevat myös katkojen pituus hyötylaskennassa (*pause-time*) ja muistin kierrätyksen tehokkuus (*promptness*). Yleensä optimointi tehdään läpäisykyvyn suhteen, mutta esimerkiksi mobiileissa laitteissa optimoinnin kohteena on usein muistinkäyttö. Muistinkäyttö ei ole sama kuin ohjelmalle varattu muisti, vaan se tarkoittaa samaan aikaan aktiivisessa käytössä olevaa muistia, ohjelman jalanjälkeä (*footprint*), jonka koko vaikuttaa muistinhallinnan kustannuksiin monella tavalla. Joskus joudutaan vähentämään roskienkeruun aiheuttamien katkojen pituutta. Katkojen pituus vaikuttaa negatiivisesti esimerkiksi interaktiivisen grafiikkaohjelman käyttäjäkokemukseen. Toisaalta WWW-palvelimen toiminnassa ei käyttäjä erota katkoja verkkoviiveestä.

Optimoinnista aiheutuvat muistikustannukset voivat olla kolmenlaiset: kasvaneen muistin tarpeen lisäksi vähemmän ilmeisiä ovat muistin pirstoutuminen pieniksi lohkoiksi (*fragmentation*) ja niin sanottu paikallisuuden ongelma, joka liittyy viittauspaikallisuuteen (*reference locality*) ja tarkoittaa toisiinsa läheisesti liittyvien oloiden hajaantumista fyysisessä muistissa.

Keko täytyy mitoittaa siten, että muisti ei lopu silloinkaan, kun ohjelman käytössä on suurin mahdollinen määrä tarpeellisia muistitavuja (*high watermark*), mutta muistin pirstoutumisen takia pienintä mahdollista kokoa on vaikea ennustaa. Keon koko vaikuttaa kuitenkin suuresti siihen, kuinka usein joudutaan siivoamaan muistia ja kuinka pitkän katkon se aiheuttaa muuttimen toiminnassa. Ohjelman vasteaikoihin vaikuttavat myös roskienkeruun keskeytettävyys tai rinnakkaisuus. Keskeytettävyyden parantaminen vaatii yleensä uhrauksia sekä kellosyklien että jalanjäljen osalta.

3.2. Muistin varaus ja muistin pirstoutuminen

Roskienkeruumenetelmä ja muistinvarausmekanismi ovat usein vahvasti kytköksissä toisiinsa. Kerääjälle asetetut suorituskykyvaatimukset koskevat myös muistinvarauksen ja vapautuksen tehokkuutta, lisäksi kerääjällä voi olla vaatimuksia olioiden asettelun suhteen muistissa. Muistinvarauspolitiikka ja roskienkeruumekanismi vaikuttavat suuresti siihen, kuinka pahasti muisti pirstoutuu muuttimen toiminnan vuoksi. Muistin pirstoutumisella (*memory fragmentation*) tarkoitetaan tilaa, jossa käyttämätöntä muistia on olemassa, mutta vaikeasti tai ei ollenkaan saatavilla, eli muuttimen kannalta joko käyttökelvottomassa muodossa tai näkymättömänä. Muisti voi olla pirstoutunut kahdella eri tavalla, sisäisesti ja ulkoisesti.

Ulkoisesta pirstoutumisesta (*external fragmentation*) puhutaan, kun vapaa muisti on muuttimen toiminnan takia hajaantunut pieniksi paloiksi pitkin kekoa. Muistin ulkoinen pirstoutuminen voi vaikeuttaa suuresti muistinvaraajan työtä, sillä varatessaan muistia se joutuu läpikäymään suuria tietorakenteita, jotka sisältävät tietoa vapaista muistilohkoista ja niiden koosta. Vaikka pirstoutuneessa muistissa saattaa olla paljonkin vapaata tilaa jäljellä, niin muistinvarausoperaatiot voivat epäonnistua, koska sopivan kokoista yhtenäistä lohkoa ei enää löydy.

Paljon käytetty muistinvarausmekanismi on ensimmäinen sopiva -algoritmi (*first-fit*), joka ylläpitää niin kutsutta vapaalistaa (*free-list*) vapaista muistilohkoista. Listasta etsitään ensimmäinen lohko, joka on riittävän iso tyydyttämään muistinvarauspyyntöä. Lohko jaetaan kahtia, jos se on liian iso, ja liitetään ylijäävä osa takaisin vapaalistaan. Muistin ulkoista pirstoutumista voidaan vähentää pitämällä muistilohkot osoitejärjestyksessä, jolloin vierekkäisiä lohkoja voidaan tarpeen tullen sulattaa yhteen (*coalesce*), tai sitten lohkojen koon mukaisessa kasvavassa järjestyksessä, jolloin voidaan helposti varata kooltaan lähinnä oleva vapaa lohko.

Muistin sisäiseen pirstoutumiseen (*internal fragmentation*) on syynä ainoastaan muistinvaraajan toiminta. Niin sanottu *buddy system* -varausmekanismi käsittelee tehokkuuden nimissä ainoastaan lohkoja, joiden koko on $t \times n$ tavua ($t, n \in \mathrm{N}$), missä n on jokin luvun 2 potenssi. Varattaessa pyydettyä s tavua, varataan todellisuudessa s' kokoinen lohko, missä $s' = \lceil s / n \rceil \times n$ ($s, s' \in \mathrm{N}$)[2]. Tällöin jää lohkon sisään (vähintään) $s'-s$ tavua käyttämätöntä muistia, joka on kuitenkin merkitty varatuksi.

Tavuja saatetaan menettää myös silloin, kun joudutaan asettelemaan joitakin tietotyyppejä, esimerkiksi osoittimia, 4:llä tai 8:lla jaollisiin osoitteisiin.

[2] Merkintä $\lceil\ \rceil$ tarkoittaa kattofunktiota. Funktion $\lceil x \rceil$ tulos on pienin sellainen kokonaisluku, joka on suurempi tai yhtäsuuri kuin x.

Sisäinen pirstoutuminen on ulkoista ennustettavampi ilmiö ja sille voidaan usein laskea tai jopa asettaa yläraja. Sisäisen pirstoutumisen vähentäminen vaatii myönnytyksiä joko varausalgoritmin nopeuden tai ulkoisen pirstoutumisen suhteen.

Johnstone ja Wilson [1998] tutkivat muistin pirstoutumista erilaisia muistinvarausmekanismeja käytettäessä 8 erilaisen C ja C++-kielisen ohjelman avulla. He totesivat, että muistin pirstoutumista ei käytännössä esiinny merkittävästi tai se ei vähennä suorituskykyä merkittävästi. Toisaalta Bacon ja kumppanit [2003] toteavat näistä tutkimustuloksista, että kyseiset testiohjelmat ovat kaikki eräajotyyppisiä ohjelmia, eikä niitä suoritettu palvelinohjelmien tavoin pitkään ja vaihtelevin kuormituksin.

Kannattaa muistaa, että manuaalinen muistinhallinta on melko voimaton muistin pirstoutumisongelman edessä, mutta jatkossa nähdään, että monella roskienkeruumenetelmällä on muistin pirstoutumista vähentävä tai poistava vaikutus.

3.3. Paikallisuusongelma

Paljon muistia käyttävissä sovelluksissa välimuistiosumat ja vähäinen sivutustarve ovat tehokkuuden avaintekijöitä. Joitakin poikkeuksia lukuunottamatta roskienkeruu koskettaa ainakin jossakin vaiheessa ohjelman muistia kokonaisuudessaan, kun jokaisen muistisolun tarpeellisuutta tutkitaan. Tällöin on tärkeää, että toisiinsa liittyvät oliot jakautuvat ryhmänä mahdollisimman harvoille muistisivulle. Tämä päätee erityisesti silloin, kun ohjelma ei ole täysin muistinvarainen, vaan osa muistisivuista on häädetty tukimuistiin, yleensä kovalevylle. Tällöin hajaantuneen graafin läpikäynti voi aiheuttaa jatkuvaa sivutusta.

Kun virtuaalimuistia on käytössä, viittausten hajaantumisesta voi koitua haittaa, vaikka ohjelma olisikin kokonaan muistinvarainen. Virtuaalimuistin osoitteet kuvataan fyysiseen muistiin monimutkaisten muunnostaulutietorakenteiden avulla. Jotta muunnokset kävisivät mahdollisimman nopeasti, suorittimissa on assosiatiivinen osoitemuunnosvälimuisti (*TLB, Translation Lookaside Buffer*, myös *ATC, Address Translation Cache)*, johon mahtuu muunnostietorakenteista yleensä vain osa. Olioiden hajaantuminen muistissa vähentää TLB-osumien määrää.

Roskienkeruu sellaisenaan ei suinkaan ole väli- ja virtuaalimuisti-ystävällistä toimintaa. Automaattinen muistinhallinta saattaa varata huomattavasti enemmän sivuja kuin mitä sovellus muuten tarvitsisi. Lisäksi läpikäyntivaiheessa kerääjä koskettaa olioita virtuaalimuistihallinnan kannalta satunnaisessa järjestyksessä, myös sellaisia, jotka ovat levylle häädetyillä sivuilla. Tämä aiheuttaa ylimääräistä sivutusta ja heikentää virtuaalimuistin

tehokkuutta sotkemalla muuttimen toiminnasta kerättyä dataa. Ongelmaan tartutaan yleensä tiivistämällä ohjelman jalanjälkeä siten, että se mahtuu fyysiseen muistiin.

Luvussa 7 esiteltävät ikäpohjaiset algoritmit toimivat hyvin yhteen virtuaalimuistin kanssa, koska ne uudelleenkäyttävät vaatimattoman kokoisia muistilohkoja tiheään tahtiin. Wilson ja kumppanit [1992] esittivät ensimmäisenä, että vastaavanlaista hyötyä voisi saada tiivistämällä ohjelman työmuisti (*working set*) siten, että se mahtuisi välimuistiin. Roskienkeruun ja välimuistin suhde oli 90-luvulla tutkimuksen kohteena, erityisesti edellä mainitut Wilson ja kumppanit sekä Zorn [1991] tutkivat niin kutsuttuja aggressiivisia kerääjiä, jotka toimivat edellä esitetyllä periaatteella. Reinhold [1994] argumentoi konseptia vastaan osoittamalla melko yleisesti, että roskakerätyt ohjelmat toimivat välimuistin suhteen tehokkaasti myös ilman roskienkeruuta sekä muistuttamalla väli- ja virtuaalimuistin välisistä olennaisista eroista, muun muassa siitä, että sivutuksen hinta on suhteettoman paljon suurempi kuin epäonnistunneen välimuistihaun.

Paikallisuusongelma ei rajoitu ainoastaan roskienkeruuseen vaan koskee myös hyötylaskentaa. Ongelmaa pienentävä roskienkeruualgoritmi parantaa samalla ohjelman läpäisykykyä.

3.4. Muistin kierrätys

Yleisesti arvioidaan, että keskiverto-ohjelmassa noin 95% uusista olioista ei selviä ensimmäisestä roskienkeruusta. Havainnolla on merkitystä vähittäisten menetelmien kannalta muistin kierrätyksen tehokkuuden suhteen. Näiden hyötylaskennan lomassa askeleittain etenevien algoritmien tehokkuutta voidaan nimittäin ilmaista sillä, kuinka paljon muistissa on jäljellä roskia roskienkeruun päätyttyä, eli kuinka hyvin algoritmi selviää roskienkeruun aikana syntyneestä niin sanotusta kelluvasta roskasta (*floating garbage*).

Kelluvan roskan lisäksi muistin kierrätyksen tehokkuuteen vaikuttaa se, voidaanko lakaisu lomittaa keräykseen. Muisti kiertää tehokkaimmin, jos voidaan suorittaa varhainen lakaisu (*early reclamation*), eli roskien varaaman muistin kierrättäminen voi alkaa jo ennen keräyksen päättymistä.

3.5. Tehokkuuden mittareista

Kuten luvun alussa todettiin, roskienkeruulla on kolmenlaisia kustannuksia: aika- ja tilakustannukset sekä hyötylaskennan katkot. Muistinhallintatutkimuksessa aika mitataan usein allokoidulla muistimäärällä. Siten roskienkeruun käsittelemä (koskettama, kopioima, yms.) muistimäärä jaettuna ohjelman allokoimalla muistilla on vertailukelpoinen aikakustannusmittari. Tätä kustutaan *mark-cons-ratio* –mittariksi (*mark-and-sweep* + *cons*, jossa *cons* on LISP-kielen

allokointioperaatio). Algoritmeilla voi lisäksi olla muita ylimääräisiä laskentakustannuksia, kuten sivutus-, kirjoitusmuuri- ja synkronointikustannukset sekä olioiden siirtelystä johtuvat välimuistihutikustannukset. Ohjelman kokonaissuoritusaika, joka sisältää edellisten lisäksi myös hyötylaskennan, voi myös olla kelpo mittari verrattaessa eri algoritmeja saman ohjelman yhteydessä.

Tilakustannusten osalta kokonaismuistimäärää kiinnostavampi mittari lienee jalanjälki, se osa kekoa, joka on kerralla muuttimen ja kerääjän käytössä. Pieni jalanjälki tietää yleensä tehokkaampaa muistinhallintaa (esimerkiksi vähemmän sivutusta).

Kerääjän tekemä työ roskienkeruun aikana määrää katkon pituuden muuttimen toiminnassa. Tehtyä työtä on tapana mitata käsiteltyjen tavujen lukumäärän ja keon koon suhteena. Kiinnostavia tunnuslukuja ovat katkojen keskimääräinen pituus sekä pisimmän mahdollisen tauon pituus.

Ajallisesti roskienkeruun tehoa mitataan yleensä käänteisesti, muuttimen käyttöasteen (*mutator utilization*) avulla. Muuttimen käyttöaste saadaan mittaamalla muuttimen hyödyntämä aika tietyn aikavälin (*time window*) sisällä. Rajattu muuttimen käyttöaste (*bounded mutator utilization*) on vähimmäiskäyttöaste määrätyn pituiselle tai sitä pitemmille aikaväleille.

4. Klassiset algoritmit

Roskienkeruun aihepiiristä on kirjoitettu monta katsausartikkelia kuluneiden vuosikymmenien aikana. Tunnetuimmat lienevät peräisin Cohenilta [1981] ja Wilsonilta [1992]. Tämän luvun klassisten algoritmien esittely pohjautuu pitkälti Wilsonin artikkelin sisältöön täydennettynä artikkelin jälkeisten saavutusten kertaamisella.

4.1. Viittauslaskenta

Viittauslaskureita käyttävä tekniikka voi täyttää jopa reaaliaikaisuuden vaatimuksia, sillä se toimii hyötylaskennan lomassa, eikä yleensä kaappaa itselleen laskentaa pidentäen vasteaikoja. Tämä ei kuitenkaan tarkoita, että se ei vaatisi paljon järjestelmän voimavaroja. Viittausmuuttujaan tehty muutos aiheuttaa kahden ylimääräisen aritmeettisen operaation suorittamisen (vähennys, lisäys), sillä joudutaan päivittämään kahden olion viittauslaskuria. Samalla kun jonkin olion viitelaskuri tulee nollaksi, täytyy vähentää kaikkien niiden olioiden laskureita, joihin kyseinen olio viittaa. Jos olio on suuren puurakenteen juuri, hyötylaskenta voi keskeytyä pitkäksi aikaa. Avainasemassa olevan olion viittauslaskurin muuttuminen nollaksi voi propagoitua todella pitkälle transitiivisessa sulkeumassa ja aiheuttaa paljon kuolemia (ks. luku 8). Ratkai-

suna tähän on erillisen listan ylläpito suurista tietorakenteista, jotka ovat vapautettavissa. Lista tyhjennetään kun se ei haittaa muuttimen toimintaa, tai viimeistään silloin kun muisti loppuu.

Viitelaskurin varaama tila voi olla myös varteen otettava tekijä. Vaikka bitti tai tavu riittäisi useimpiin käytännön tilanteisiin, joskus joudutaan varaamaan suurempi kokonaisuus arkkitehtuurisista syistä. Jos ohjelma käyttää todella paljon pienikokoisia dynaamisesti varattuja olioita (esim. pieniä kokonaislukuja), viitelaskureiden osuus ohjelman käyttämästä muistista voi lähentyä 50 prosenttia. Jos viitelaskurille varataan yksi ainoa tavu, ylivuodon vaara kasvaa. Ylivuodon sattuessa laskuria ei voi enää käyttää, vaan olio jää keräämättä.

Viitelaskuritekniikkaa ei juurikaan käytetä niissä ohjelmointikielissä, jotka sallivat syklisiä rakenteita. Nimittäin algoritmin roskakriteeristä “jos olioon ei viitata, se on roskaa” ei seuraa, että “jos olioon viitataan, se ei ole roskaa”. Niin kauan kuin kaksi oliota viittaavat toisiinsa, kummankaan viitelaskuri ei tule nollaksi, silti molemmat oliot ovat roskia, jos kumpaankaan ei ole olemassa polkua juurijoukosta.

Tekniikkaa käytettäessä yleensä joudutaan ottamaan jokin sopiva olioverkkoa läpikäyvä menetelmä avuksi, jos halutaan

- eheyttää muistia, tai
- kerätä ne oliot, joiden viitelaskuri on vuotanut yli, tai
- poistaa sykliset rakenteet muistista.

Viittauslaskureita käyttävää tekniikkaa suositaan puhtaasti funktionaalisissa ohjelmointikielissä, sillä ne eivät salli syklisiä rakenteita, ja voivat hyödyntää viitelaskuria "vain yhdestä paikasta viitattujen" (*unique reference*) omainaisuuksien tarkistamiseen.

Viittauslaskentatekniikalla on siis kaksi heikkoutta, syklisistä rakenteista johtuva epätäydellisyys sekä työläys, jonka aste on verrannollinen muuttimen tekemän työn määrään. Klassinen tapa vähentää viittauslaskennan kustannuksia on viivytetty viittauslaskenta (*deferred reference counting*), jonka esittivät Deutsch ja Bobrow [1976]. He havaitsivat, että suurin osa viittauksista suuntautuu paikallismuuttujista (pinosta) kekoon. Siksi nämä lyhytikäiset ja tiheään muuttuvat viittaukset kannattaa jättää kirjanpidosta pois ja keskittyä keko-olioiden välisiin viittauksiin. Tällöin olion viittauslaskurin nollautuminen ei tarkoita enää automaattisesti, että olio olisi kuollut, vaan laskureita joudutaan aika ajoin päivittämään tutkimalla kaikkia pinosta lähteviä viittauksia. Deutschin ja Bobrowin menetelmä tallentaa tällaisten olioiden osoitteet niin sanottuun nollatauluun (*ZCT, Zero Count Table*). Samalla kuin pinoa tutkitaan, taulusta poistetaan ne oliot, joihin löytyy viittaus, siten jäljelle

jäävät ovat kuolleet ja lakaistavissa. Menetelmä jakaa siis ohjelman suoritusajan kahteen jaksoon: muutinjakson (*deferred phase*) aikana ohitetaan ja viittauslaskentajakson (*RC phase*) aikana lisätään paikallismuuttujaviittaukset laskureihin.

ZCT-taulut voidaan korvata puskuroimalla laskurimuutokset ja suorittaa ne viittauslaskentajakson alussa. Bacon ja Rajan [2001] korvaavat rinnakkaisessa menetelmässään ZCT-taulun väliaikaisin lisäyksin ja vähennyksin (*temporary increment/decrement*), jotka puskuroidaan erikseen kilpailutilanteiden välttämiseksi. Viittauslaskentajakson alussa paikallismuuttujat käydään läpi ja tehdään tarvittavat lisäykset laskureihin (ennen puskuroituja laskurimuutoksia). Vaiheen päätyttyä tehdään sama asia toisin päin, eli vähennetään pinoviittaukset laskureista.

Levanoni ja Petrank [2001] ovat todennet, että viivytetyn viittauslaskennan periodisuus tekee muutinvaiheen osoitteenmuutokset viimeistä muutosta lukuunottamatta merkityksettömiksi. Toisin sanoen ainoastaan osoittimen alkuperäisellä ja lopullisella arvolla on merkitystä, muut arvot voidaan unohtaa, tai "sulattaa" viimeiseen arvoon. Tämä vaatii kuitenkin sen, että alkuperäiset viittaukset tallennetaan johonkin, jotta vertaaminen lopullisiin arvoihin olisi mahdollista.

Syklisten rakenteiden ongelma on ollut luultavasti suurin este viittauslaskennan yleistymiselle nykyisissä automaattisen muistinhallinan suoritusjärjestelmissä. Ongelmaan on jo pitkään haettu ratkaisua. Ensimmäisen viittauslaskennan lomassa toimivan syklisten rakenteiden keruualgoritmin julkaisi Christopher [1984]. Algoritmi etsii kokeilevasti roskasyklejä (*garbage cycle*), eli irrallisia syklisiä rakenteita vähentämällä väliaikaisesti tiettyjen olioiden viittauslaskuria. Tätä tehdään siinä toivossa, että muutokset propagoituvat syklisiin tietorakenteisiin, joiden viittauslaskurit nollautuvat, jos ne ovat irrallisia. Martínez ja kumppanit [1990] tarkensivat menetelmää ja havainnollistivat sitä kolmivärimerkinnän avulla. Algoritmi pohjautuu kahteen seuraavaan perustavaa laatua olevaan havaintoon roskasykleistä:

- Roskasykli voi syntyä ainoastaan silloin, kun arvoltaan yhtä suurempaan viittauslaskuriin tehdään vähennys. (Lisäys ei voi synnyttää roskia, nollautuminen synnyttää havaittua roskaa.)
- Roskasyklin olioiden viittauslaskurit koskevat ainoastaan syklinsisäisiä viittauksia, siksi näiden sisäisten viittausten vähentäminen paljastaa koko syklin roskaksi.

Menetelmä pitää potentiaalisena roskasyklijuurena jokaista sellaista oliota, jonka viitelaskuria vähennetään yllämainitulla tavalla, ja tutkii tällaisesta juuresta lähtevää transitiivista sulkeumaa syvyyshaulla. Lins [1992] tehosti menetelmää viivyttämällä roskasyklien etsintää keräämällä juuret listaan

myöhempää tutkimusta varten (*lazy sweeping*). Tämä suodattaa pois ne juuret, joiden viittauslaskuri lopulta nollautuu ja estää samojen juurien toistuvaa tutkimista.

Bacon ja Rajan [2001] kehittivät Linsin ideaa eteenpäin tutkimalla kaikkia juuria samalla haulla. Menetelmä toimii (pähkinänkuoressa) seuraavasti: Algoritmi kerää potentiaaliset juuret listaan viittauslaskentajakson aikana ja merkkaa viitatut oliot puskuroiduksi duplikaattien välttämiseksi. Seuraavaksi algoritmi lähtee kulkemaan juurista lähteviä polkuja pitkin värjäten polun varrella olevia olioita harmaaksi ja vähentäen niiden viittauslaskuria yhdellä. Algoritmi etenee graafissa aina mahdollisimman syvälle, joko umpisolmuun (solmuun, josta ei pääse enää eteenpäin) tai harmaan solmuun, joista se joutuu kääntymään takaisin. Kun kaikkien juurien kaikki polut on käyty läpi, ne harmaat solmut, joiden viittauslaskuri on nolla, ovat roskasylien elementtejä ja ovat lakaistavissa. Toisin kuin Linsin algoritmi, joka on pahimmillaan neliöinen, toimenpide onnistuu aina pahimmassakin tapauksessa ajassa O(N + E), missä N ja E ovat solmujen ja kaarien lukumäärät.

Vaikka syklisten rakenteiden purku onkin ratkaistavissa, viittauslaskenta ei ota kantaa muistin pirstoutumisen ja viittauspaikallisuuden ongelmiin. Mikään tunnettu muunnelma ei voi siirrellä olioita paikasta toiseen, joka on perinteinen tapa vähentää kyseisiä haittoja.

4.2. Merkitse ja lakaise (mark-and-sweep)

Tekniikkana mark-and-sweep on yksinkertainen ja sen periaate on täysin virheetön. Verkon läpikäynti voidaan suorittaa leveyssuunnassa (leveyshaulla), edeten olioiden etäisyyden mukaan juurijoukosta (*breadth-first*), tai pysty-suunnassa (syvyyshaulla), edeten graafissa aina niin syvälle kuin mahdollista (*depth-first*). Muiden muassa Moon [1984] on todennut, että etenemistavalla voi olla suurtakin merkitystä viittauspaikallisuuden kannalta. Läpikäydyt oliot merkitään joko bittikarttaan, tai merkintä tehdään olion yhteyteen varattuun tavuun (*tag*). Bittikartan käyttö on huomattavasti välimuistiystävällisempi tapa, sillä se ei muuta olion yhteyteen varatun varsinaisen muistin sisältöä.

Lopuksi lakaistaan muistia keräämällä kaikki merkitsemättömät oliot muistinvaraajan vapaalistaan. Algoritmia voidaan tehostaa jättämällä lakaisuvaihe pois, ja suorittaa se vähittäisesti muistin varaamisen yhteydessä (*lazy sweeping* tai *mark-and-don't-sweep*). Hieman samaan tapaan toimii niin kutsuttu polkumyllytekniikka (*treadmill collection*), jonka esiti Baker [1992]. Algoritmi jakaa keon kahteen loogiseen osaan ylläpitämällä kahteen suuntaan linkitettyä listaa tarpeellisista olioista (aktiivinen lista) ja vapaista muistipaikoista (passiivinen lista). Kun vapaita muistipaikkoja ei ole enää saatavissa tai vapaa muisti on pirstoututunut liian pieniksi paloiksi, tehdään

roskienkeruu. Jäljellä olevat vapaat muistipaikat liitetään aktiiviseen listaan, näin passiivinen lista tyhjenee. Samalla kun olioverkkoa käydään läpi, tarpeelliset oliot siirretään passiiviseen listaan. Roskienkeruun päätyttyä aktiivinen lista sisältää roskat ja varaamattomat muistialueet. Vaihtamalla listojen roolit (*flip*), päästään alkutilanteeseen. Tekniikka ei ota kantaa muistin pirstoutumiseen ja sen muistintarve on listojen toteutuksesta riippumatta suuri.

4.3. Merkitse ja tiivistä (mark-compact)

Tiivistävä (*compressing*) muunnelma mark-and-sweep -algoritmista on kehitelty muistin pirstoutumisen välttämiseksi ja vähentämään paikallisuusongelmaa. Algoritmi kerää roskien sijasta tarpeelliset oliot. Merkitseminen suoritetaan tavalliseen tapaan, mutta siivousvaiheessa algoritmi siirtää merkityt oliot vierekkäin muistin alkupäähän. Tällä tavoin saavutetaan toisiinsa liittyvien olioiden läheinen fyysinen sijainti muistissa.

Algoritmin todellinen teho piilee siihen liittyvässä muistinvarausmallissa: kekoa voidaan käyttää pinomaisesti allokaatio-osoittimen (*allocation pointer*) avulla, joka jakaa muistin käytettyyn ja vapaaseen osaan. Uusille olioille ei tarvitse etsiä sopivaa vapaata lohkoa, vaan ne luodaan peräkkäin käytetyn alueen jatkeeksi siirtäen osoittimen aina eteenpäin. Tiivistettäessä olioiden uuden sijainnin laskeminen, niiden varsinainen siirtely ja viittausmuuttujien päivitys ovat kuitenkin varsin kalliita operaatioita.

4.4. Pysäytä ja kopioi (stop-and-copy)

Tiivistävän algoritmin ideaa edelleen kehittävä tiivistäen kopioiva (*copying-compacting*) algoritmi vähentää edellisen laskentakustannuksia suorittamalla tiivistämisen samalla, kun käy olioverkkoa läpi. Ohjelman suorituksen aikana muisti on jaettu kahteen samankokoiseen osaan – puoliavaruuteen (*semispace*) – joista toinen on ohjelman käytössä ja toinen on kopiointivara (*copy reserve*). Kun käytössä oleva puolisko – lähdeavaruus (*from-space*) – varataan täyteen, alkaa roskienkeruu, jossa tarpeelliset oliot kopioidaan tiivistäen toiseen muistipuoliskoon – kohdeavaruuteen (*to-space*). Roskienkeruun päätyttyä puoliskoiden roolit vaihtuvat (*flip*). Varsinaista lakaisua ei tarvita (*en block/mass reclamation*).

Cheney [1970] esitti tiivistäen kopioivan algoritmin, joka käyttää leveys-suuntaista läpikäyntiä. Kopiointi etenee aaltoina: kopioituaan ensin juuri-joukosta saavutettavat oliot algoritmi käy kopioiden viittausjäsenmuuttujat läpi, mistä syntyy uusi kopiointiaalto. Heti kun viittausmuuttujan osoittama olio on kopioitu, muuttuja päivitetään osoittamaan olion uuteen paikkaan. Syklisten rakenteiden takia algoritmi käyttää viittausten uudelleenohjaus-tekniikkaa (*memory forwarding*). Jokaiseen kopioituun olioon lähdeavaruudessa

(*broken heart*) jätetään niin sanottu kopio-osoitin olion uuteen paikkaan (*forwarding reference*). Mikäli kopioidusta oliosta viitataan jo aikaisemmin kopioituun olioon, viittausmuuttujan arvo päivitetään vastaamaan tätä osoitinta.

Tiivistäen kopioiva algoritmi ehkäisee olioiden hajaantumista ja säilyttää muistin eheyden huomattavasti pienemmin kustannuksin kuin tiivistävä algoritmi, sillä keruun kesto on suhteessa tarpeellisen oliojoukon kokoon eikä keon kokoon. On kuitenkin selvää, että tiivistäen kopioivaa algoritmia käyttävää roskienkeruuta joudutaan suorittamaan useammin, sillä muistia on käytettävissä puolet vähemmän.

5. Vähittäinen roskienkeruu

Vähittäisten menetelmien ideana on ehkäistä pitkät tauot hyötylaskennassa tekemällä työtä lyhyitä aikoja kerrallaan. Tarkoituksena on ohjelman vasteaikojen parantaminen, haasteena taas keruun riittävän nopean edistymisen varmistaminen. Ajettaessa reaaliaikaisuuteen pyrkiviä sovelluksia roskienkeruun rinnakkaisuus, vähittäisyys tai keskeytettävyys nousevat kynnyskysymyksiksi. Hyötylaskennan lomassa toimivalla viitelaskuritekniikalla on reaaliaikaisuuden vaatimat ominaisuudet, mutta muiden negatiivisten ominaisuuksiensa takia tekniikkaa sovelletaan harvoin. Jäljelle jäävät siis mark-and-sweep ja stop-and-copy -tekniikoiden optimoidut muunnelmat sekä osittainen roskienkeruu, joka jakaa keon itsenäisesti kerättäviin, sopivan kokoisiin osioihin.

5.1. Vähittäin etenevä roskienkeruu

Perusmuodossaan mark-and-sweep ja stop-and-copy -tekniikat ovat stop-and-go -tyyppisiä, joten ne vaativat muuttimen toiminnan jäädyttämistä roskienkeruun ajaksi. Algoritmien vähittäisissä muunnelmissa ongelmat ilmenevät merkintävaiheessa, sillä muutin saattaa muuttaa graafia kesken läpikäynnin. Roskienkerääjältä kuitenkin vaaditaan, että huolimatta voimakkaasta häirinnästä se toimii turvallisesti (virheettä), haittaamatta muuttimen toimintaa. Tilanne on erityisen hankala tiivistävien ja kopioivien algoritmien osalta. Asetelmaa voidaan verrata monen lukijan ja monen kirjoittajan ongelmaan, sillä sekä muunnin että roskienkerääjä voivat muuttaa graafia. Tämäntapainen toiminta vaatii raskasta koordinointia.

Dijkstran ja kumppaneiden [1978] kolmivärimerkintä (*tricolor-marking*) on omiaan havainnollistamaan ongelmaa. Kolmella värillä – musta, harmaa, valkoinen – merkitystä graafista on nähtävissä kunakin hetkenä läpikäynnin tila. Läpikäynnin alussa graafin kaikki solmut ovat valkoisia. Edetessään

graafissa roskienkerääjä värjää koskettamansa solmut harmaiksi. Joutuessaan umpisolmuun, se värjää solmun mustaksi. Mustaksi muuttuvat myös ne solmut, joista pääsee vain mustiin solmuihin. Läpikäynnin päättyessä kaikki alkiot ovat joko mustia (tarpeellisia), tai valkoisia (roskia). Tässä muodossaan algoritmi ei juuri eroa edellä esitetyistä, joiden voidaan oikeastaan katsoa käyttävän kaksivärimerkintää. Harmaa väri on kuitenkin tarpeen, sillä se ilmaisee, että solmun edustama olio on tarpeellinen, mutta (kaikista) sen jälkeläisten tarpeellisuudesta ei ole vielä tietoa.

Keskeytyksen jälkeen roskienkerääjä jatkaa työtään harmaista solmuista eteenpäin palaamatta enää mustiin solmuihin, vaikka muutin olisikin tehnyt niistä tarpeettomia ennen merkintävaiheen päättymistä. Tämä ei kuitenkaan riko virheettömyysvaatimusta, sillä tällaiset oliot tulevat kerätyksi seuraavan roskienkeruun yhteydessä. Asialla on korkeintaan tehokkuusmielessä merkitystä.

Tilanne on toinen, jos muutin hävittää kaikki viittaukset valkoiseen solmuun, mutta samalla luo uuden viittauksen siihen mustasta solmusta. Tällöin valkoinen solmu tulisi kerätyksi tarpeellisuudestaan huolimatta. Kuten edellisestä esimerkistä on nähtävissä, virheettömyysvaatimus edellyttää, että mustasta solmusta ei voi päästä valkoiseen solmuun suoraan. Kyseessä on niin sanottu invarianttisääntö: musta solmu on invariantti, jos harmaiden solmujen vyöhyke erottaa sen valkoisista solmuista.

5.2. Laskennan ja roskienkeruun synkronointi

Invarianttisäännön voimassapitoon on tarjolla kaksi mekanismia, lukumuuri (*read-barrier*) ja kirjoitusmuuri (*write-barrier*). Muurit ovat synkronointivälineitä, joiden tehtävänä on pitää kerääjä ajan tasalla niistä muuttimen tekemistä muutoksista, joilla on keruun kannalta merkitystä. Kerääjän ja muuttimen näkemys graafista voivat siis poiketa toisistaan: kerääjä voi pitää kuolleita olioita elävinä, mutta ei toisin päin. Tällaisia kerääjiä sanotaan konservatiivisiksi (*conservative*).

Lukumuurin käyttö tarkoittaa käytännössä, että muuttimen lukiessa valkoisen solmun osoitteen solmu värjätään harmaaksi. Näin ollen muutin ei voi luoda viittauksia valkoisiin solmuihin. Kirjoitusmuuria voidaan käyttää kahdella tavalla, tarkkailemalla joko mustiin tai muunvärisiin solmuihin tehtyjä muutoksia.

Molemmissa tapauksissa ohjelma suorittaa kirjoitusoperaation yhteydessä ylimääräisiä käskyjä, joiden tarkoitus on pitää roskienkerääjän tiedot ajan tasalla. Nämä käskyt voidaan lisätä ohjelmaan jo käännösaikana. Kirjoitusmuureista koituu yleensä vähemmän kustannuksia kuin lukumuureista, sillä kirjoitusoperaatioita suoritetaan vähemmän.

5.3. Kirjoitusmuurit

Klassinen tapa hyödyntää kirjoitusmuuria on roskienkeruun aikana hyötylaskennan muuttamien viittausten vanhojen arvojen tallentaminen (*snapshot-at-beginning* [Wilson, 1992][3]). Läpikäynnin päätyttyä roskienkerääjä aloittaa uuden läpikäynnin käyttäen tallennettuja osoittimia (*remembered set*, *remset*) toisena juurijoukkona (*secondary roots*). Tämä varmistaa sen, että mikään olio ei häviä roskienkerääjän näkyvistä, ja sen tiedot pysyvät näin ajan tasalla. Asia voidaan toki nähdä myös konkreettisena haittana: algoritmin tehoaste jää alhaiseksi, sillä läpikäynnin aikana syntyneitä roskia ei lakaista.

Kelluvan roskan kannalta tehokkaampi tapa on tarkistaa, onko viittausmuuttujan päivitys tehty mustaan solmuun. Jos näin on asianlaita, joko musta, tai viitattu solmu väritetään harmaaksi. Tällä kirjoitusmuurin käyttötavalla (*incremental update*) on omana erikoisuutenaan se, että uudet oliot voidaan syntyessään turvallisesti merkitä valkoisiksi, sillä vaikka uusi olio liitettäisiinkin mustaan solmuun, jompikumpi solmuista muuttuu harmaaksi [Dijkstra *et al.*, 1978]. Muussa tapauksessa solmu jää valkoiseksi, kunnes läpikäynti (mahdollisesti) koskettaa sitä. Asialla on merkitystä olioiden lyhytikäisyyshavaintoa ajatellen, sillä keruun aikan syntyneet roskat voidaan siivota.

Kirjoitusmuuria voidaan käyttää varmistamaan myös kopioivien algoritmien virheettömyys. Muurin tehtävänä on toistaa lähdeavaruuden olioihin tehdyt muutokset kohdeavaruudessa. Koska kirjoitusoperaatiot ovat kalliita, tätä menetelmää käytetään vain erikoistapauksissa.

5.4. Lukumuurit

Cheneyn [1970] tiivistäen kopioiva algoritmi on helposti muutettavissa vähittäiseksi lukumuurin avulla Bakerin [1978] esittämällä tavalla: Lukumuurin tehtävänä on varmistaa, että muutin työskentelee aina olion "oikean" version kanssa. Jos olio on jo kopioitu toiseen muistipuoliskoon, lukumuuri kääntää kaikki viittaukset sinne. Muussa tapauksessa kopiointi tehdään saman tien. Tämä vastaa solmun värjäämistä harmaaksi. Uudet oliot luodaan valmiiksi kopioinnin kohdeavaruuteen (mustiksi solmuiksi), mikä johtaa siihen, että ne selviävät roskienkeruusta huolimatta mahdollisesta tarpeettomuudestaan roskienkeruun päätyttyä. Tosin roskienkeruun aikana roskaksi muuttuneet vanhat oliot siivotaan.

Lukumuuria voidaan käyttää myös mark-and-sweep -tekniikan kanssa, erityisesti polkumylly-algoritmin yhteydessä. Syklisesti yhteen linkitetty lista sisältää järjestyksessä neljä joukkoa: passiivinen lista (vanhat oliot), aktiivinen

[3] Ainoastaan nimitys "snapshot-at-beginning" on peräisin Wilsonilta.

lista, uudet oliot ja vapaat muistialueet. Aktiivinen lista ja uusien olioiden joukko ovat aluksi tyhjiä. Joukkojen erottamiseksi on käytössä neljä osoitinta. Läpikäynnin aikana passiivisen listan solmut siirretään aktiiviseen listaan harmaaksi väritettyinä. Uudet oliot syntyvät mustina solmuina, eli ne selviävät roskienkeruusta. Lukumuurin aktivoituessa luettu olio siirretään aktiiviseen listaan. Kun läpikäynti on päättynyt, vierekkäiset joukot sulautetaan yhteen: uusien olioiden lista liitetään aktiiviseen listaan ja vanhojen olioiden lista (sisältää roskat) liitetään vapaiden muistialueiden listaan.

5.5. Vähittäisyyden tehokkuuskysymykset

Kuten luvussa neljä todettiin, optimoinnissa on aina (vähintään) kaksi vastakkaista näkökulmaa. Kun puhutaan vähittäisten algoritmien tehokkuudesta, nämä näkökulmat ovat tarvittava laskenta (*overhead*) ja syntyneiden ja lakaistujen roskien määrän suhde. Optimaalisuudesta voidaan puhua vasta, kun olemme määritelleet kahden näkökulman tärkeyssuhteen, joka on kuitenkin sovelluskohtainen asia.

Yleistäen voidaan todeta, että jos sovellus luo jatkuvasti olioita, jotka tulevat lyhyessä ajassa tarpeettomiksi, saattaa olla parempi käyttää "huolellista" algoritmia, joka tarkistaa myös uusien olioiden tarpeellisuuden. Yleistys ei aina kuitenkaan päde, sillä "huolimattomat" algoritmit vaativat vähemmän laskentaa, joten niitä voidaan suorittaa nopeammin ja useammin. Jos sovellus synnyttää pitkäikäisiä olioita, tiukat koordinointitoimenpiteet ovat vain haitaksi, ja huolimattoman algoritmin käyttö on kannattavaa.

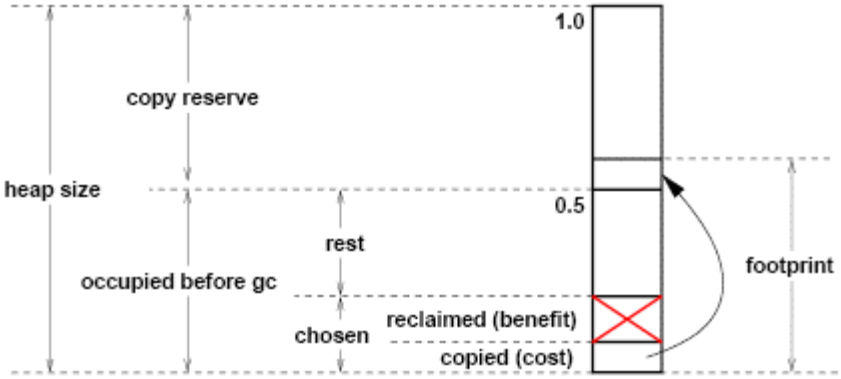
5.6. Osittaiset menetelmät

Roskienkeruun vähittäisyyteen voidaan päästä myös jakamalla kerättävien olioiden joukko alijoukkoihin ja käsittelemällä nämä osiot kokonaisuudessaan yksi kerrallaan hyötylaskennan lomassa. Yleisin osittainen menetelmä on luvussa 6 esiteltävä ikäpohjainen roskienkeruu, joka osioi olioiden iän perusteella, uusin menetelmä taas lienee yhdistävyyspohjainen menetelmä, johon tutustutaan luvussa 8.

Osiot täytyy valita sopivasti siten, että niiden välille jää mahdollisimman vähän riippuvuuksia, jotka aiheuttavat monenlaisia vaikeuksia. Osittaisten menetelmien suurin tekninen ongelma on niin sanottu "nepotismi". Nepotistinen tilanne syntyy, kun kuollut olio keräämättömässä osiossa pitää perusteettomasti hengissä olioita kerätyissä osioissa. Osittaiset kerääjät pitävät yleensä kirjaa muuttimen synnyttämistä riippuvuuksista osioiden välillä esimerkiksi kirjoitusmuurin avulla.

Osittaisen keruun toimintaperiaate voidaan nähdä kuvasta 3, joka esittää stop-and-copy –tyyppisen kerääjän osioiman keon anatomiaa. Keruu suori-

tetaan valitussa osiossa (kuvassa *chosen*), josta tarpeelliset oliot kopioidaan kopiointivaraan (kuvassa *copy reserve*). Muu lähdeavaruus (kuvassa *rest*) on kuitenkin osa keruun jalanjälkeä (kuvassa *footprint*), koska siellä saattaa olla päivitettäviä osoittimia.

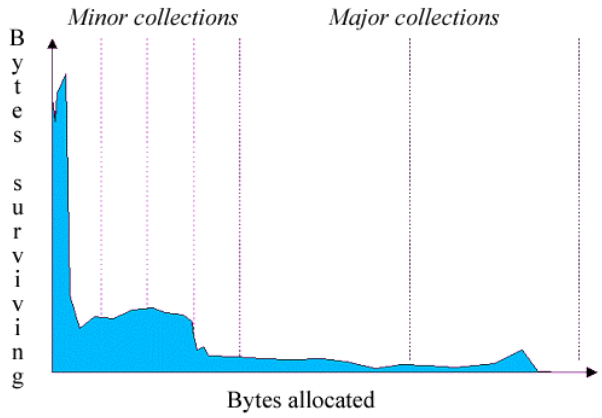


Kuva 3 Kopioivan kerääjän osioima keko [Hirzel *et al.*, 2003].

6. Ikäpohjainen roskienkeruu

Sukupolvittaisen tai ikäpohjaisen roskienkeruun idea perustuu jo aiemmin mainittuun havaintoon siitä, että suurin osa ohjelman luomista olioista on väliaikaisia, eli ne tulevat lyhyessä ajassa tarpeettomiksi, kun taas pieni osa olioista on pitempään aikaan käytössä. Ilmiö näkyy kuvasta 4, joka esittää tyypillistä olioiden ikäjakaumaa.

Ikäpohjaiset menetelmät ryhmittävät oliot sukupolviin ja huolehtivat roskienkeruusta sukupolvikohtaisesti. Mitä nuorempi sukupolvi, sitä useammin siitä kerätään roskat. Etenkin kopioivat algoritmit hyötyvät tällaisesta olioiden kategorisoinnista, sillä pitkäikäisten olioiden jatkuva kopiointi on kallista. Uudet oliot jäävät suurella todennäköisyydellä lyhytikäisiksi, niitä ei siksi tarvitse moneen kertaan kopioida. Jos taas ne saavuttavat tietyn iän, ne siirretään vanhempaan sukupolveen. Tekniikkaa voidaan verrata kullanhuuhdontaan. Tässä luvussa tarkastellaan tarkemmin, miten ikäpohjaista roskienkeruuta voi tehostaa.



Kuva 4 Tyypillinen olioiden ikäjakauma [Sun, 2003]

6.1. Olioiden ikä ja sukupolvien lukumäärä

Sukupolvittainen roskienkeruu tarjoaa mahdollisuuden valita sopiva algoritmi jokaista ikäluokkaa varten. Ajatellaan kuitenkin yksinkertaisempaa esimerkkiä, joka käyttää kopioivaa algoritmia. Muisti jaetaan esimerkiksi kolmeen osaan, joista jokainen vielä puolitetaan tiivistäen kopioivaa algoritmia varten. Kun nuorimman sukupolven, eli lastentarhan (*nursery*), aktiivinen muistipuolisko täyttyy, aloitetaan roskienkeruu tavalliseen tapaan, mutta kynnysikää saavuttaneet oliot kopioidaan keskimmäisen sukupolven aktiiviseen puoliskoon. Keskimmäinen sukupolvi siivotaan samalla tavalla, mutta harvemmin. Vanhimpaan sukupolveen päässeet oliot ovat ipso facto pitkäikäisiä.

Edellisessä esimerkissä on kuitenkin monta sellaista yksityiskohtaa, joita kannattaa miettiä tarkemmin. Eräs tällainen yksityiskohta on sukupolvien lukumäärä ja niille varattujen muistialueiden kokojen suhde. Eri sukupolville voidaan valita myös sopiva kynnysikä, lisäksi joudutaan miettimään keinoja olioiden iän tallentamiseen.

Sukupolvien lukumäärä ja kynnysikä liittyvät läheisesti yhteen: mitä suurempi määrä sukupolvia on käytössä, sitä nopeammin oliot kannattaa ylentää (*promote*). Ääritapauksessa roskienkeruusta selviävät oliot ylenevät suoraan seuraavaan sukupolveen. Edellisessä esimerkissä tämä järjestely johtaisi siihen, että vain vanhimman sukupolven muistia pitäisi puolittaa – nuorempien sukupolvien kohdeavaruus olisi seuraavan sukupolven lähdeavaruus. Tämä voi olla aika tehokas ratkaisu, mutta täytyy muistaa, että sukupolvia täytyy olla melkoinen määrä ennen kuin nuorimman ja vanhimman sukupolven oliot eroavat iältään merkittävästi.

Tehokkainta lienee kuitenkin kahtiajako, eli lastentarhan ja asettuneen sukupolven (*tenured generation*) käyttäminen. Näin lastentarhalle voidaan varata mahdollisimman suuri tila ja siten minimoidaan todennäköisyys sille, että joudutaan kopioimaan nuoria olioita. Tällöin roskienkeruu lastentarhassa (*minor collection*) kestää pitempään, vanhan sukupolven siivoaminen (*major collection*) koskettaa joka tapauksessa molempia sukupolvia.

Kuten edellä todettiin, algoritmi voidaan valita sukupolvikohtaisesti. Esimerkiksi tilan säästämiseksi voidaan käyttää merkitse ja tiivistä -algoritmia vanhimmassa tai vanhimmissa sukupolvissa. Roskienkeruu kannattaa tehdä kuitenkin vähittäin, koska roskien etsiminen vanhoista sukupolvista koskettaa yleensä kaikkia nuorempia sukupolvia. Nuorissa sukupolvissa stop-and-go -tyyppinen keruu voi olla tarkoituksenmukaisempaa, koska se joudutaan tekemään usein ja nopeasti.

6.2. Viittaukset sukupolvesta toiseen

Tähän asti emme ole kiinnittäneet mitään huomiota ikäpohjaisen roskienkeruun suurimpaan tekniseen haasteeseen, sukupolvien rajoja ylittäviin viittauksiin, joihin törmätään väistämättä. Roskia ei voi kerätä mistään sukupolvesta tuntematta kaikkia niitä viittauksia, jotka kohdistuvat muista sukupolvista kyseiseen sukupolveen. Appel [1989] on todennut, että vanhoista sukupolvista viitataan vähemmän nuorempiin sukupolviin kuin toisin päin. Viittauksista vanhempiin sukupolviin ei kannata pitää rekisteriä, koska roskien etsintä vanhoista sukupolvista koskettaa kaikkia nuorempia sukupolvia. Sen sijaan useimmissa toteutuksissa viittauksista nuorempiin sukupolviin pidetään kirjaa jollakin tavalla.

Yksi tällainen tapa on uudelleenohjaustaulukon käyttö. Sukupolven olioihin viitataan muista sukupolvista välillisesti taulukon avulla, ja taulukon alkiot lisätään roskienkeruussa juurijoukkoon. Ratkaisu ei ole kuitenkaan kovin tehokas, koska muistin lisäksi se vaatii paljon myös ylimääräisiä kellosyklejä, sillä osoittimen purku tai päivitys laukaisee aina luku- tai kirjoitusmuurin.

Pelkällä kirjoitusmuurilla pärjätään, jos tallennetaan jokaisen muuttuneen viittauksen osoite ja tarkistetaan vasta roskienkeruun yhteydessä, onko kyseessä sukupolvien rajoja ylittävä viittaus. Uudelleenohjaustaulukko on edelleen tarpeen, mutta se ei enää ole käytössä muuttimen toiminnan aikana, vaan yksinomaan roskienkerääjän toimiessa.

6.3. Ikä = tarpeellisuus?

Kaikissa tapauksissa ei edellämainittu oletus viittauksista vanhoista sukupolvista nuorempiin päde. Ajatellaan esimerkiksi suurta säiliöoliota, vaikka listaa, vektoria tai suurta taulukkoa, jonka ohjelma luo käynnistyessään

ja käyttää loppuun asti, mutta vaihtaa sen sisältöä jatkuvasti varaamalla siihen uusia olioita. Tällöin vanhasta sukupolvesta viitataan jatkuvasti nuorimpaan, joten kirjoitusmuurista koituu paljon kustannuksia ja vanhan olion kuollessa syntyy helposti roskien kierrätystä haittaava nepotistinen tilanne.

Sama pätee tietenkin sukupolvittaisen roskienkeruun perusolettamuksiin, eli siihen, että olioiden ikä ja tarpeellisuus ovat rinnastettavissa ja ohjelmat luovat paljon lyhytikäisiä olioita. Jos jostakin syystä nämä olettamukset eivät toteudu, tuhlataan sukupolvien hallinnointiin todella paljon järjestelmän voimavaroja. Tämä on erityisesti totta käytettäessä vähittäistä algoritmia.

6.4. Appel-kerääjä

Tehokkaimpiin ikäpohjaisiin menetelmiin lukeutuva Appel-kerääjä on suosittu vertauskohde benchmark-analyyseissä ja lienee klassisten menetelmien tunnetuin edustaja. Appel [1989] tehosti perinteistä ikäpohjaista mallia muuttamalla lastentarhan koon joustavaksi (*flexible-sized nursery*). Appel-kerääjä jakaa keon kolmeen osaan. Aluksi vanha sukupolvi on tyhjä ja sen koko on nolla, siten keko on jaettu tasan kopiointivaran ja käyttömuistin kesken (tässä osoitejärjestyksessä). Kun käyttömuisti tulee täyteen, suoritetaan roskienkeruu, josta eloonjääneet kopioidaan kopiointivaran alkuun vanhaksi sukupolveksi (myöhemmessä vaiheessa vanhan sukupolven jatkeeksi). Vapaaksi jäänyttä muistia puolitetaan uudestaan - kopiointivara alkaa siitä, mihin vanha sukupolvi loppuu ja käyttömuistia voi periaatteessa lajentaa varaamalla siihen lisätilaa. Koska lastentarha voi kasvaa ja pienentyä, roskienkeruun kesto voi vaihdella suuresti. Siksi Appel-kerääjälle on vaikeaa asettaa vaatimuksia hyötylaskennan katkojen suhteen.

Kopioivana menetelmänä Appel-kerääjän täytyy varmistaa, että kopiointivara on tarpeeksi suuri myös pahimmassa tapauksessa. Appel muotoilee asian invarianttisääntönä: jos ohjelman käyttämä elävä data on A, niin muistia M pitää olla vähintään $A \times 2$. Tosin M saisi olla huomattavasti suurempi kuin $A \times 2$, jotta kerääjä toimisi tehokkaasti. Elävän datan sallittua enimmäismäärää kuvaa arvo h. Edellisen invarianttisäännön ollessa voimassa h on enintään $M/2$.

Ennen pitkää vanha sukupolvi täyttyy. Tämä tapahtuu, kun vanhan sukupolven O perään kopioidaan lastentarhan N eloonjääneet N' ja $O+N'$ tulee suuremmaksi kuin arvo h. Tässä tapauksessa kopiointivara R voi siis olla pienempi kuin kopioinnin lähdeavaruus, kuitenkin varmuudella suurempi kuin $A-N'$, sillä N' on elävää dataa ja A on enintään $M/2$. Vanhan sukupolven eloonjääneet O' voidaan siis kopioida tilaan R, tilan N' perään. Roskienkeruu päättyy siihen, että N' ja O' lohkokopioidaan takaisin keon alkuun.

Oletusarvoisesti vakion h arvoksi otettaan M/3, jolloin pahimmissakin tapauksessa O = N = R.

Appel jättää artikkelissaan monta teknistä yksityiskohtaa auki. Esimerkiksi yllämainittu lohkokopiointi ilmeisesti vaatii sen, että kopioitavissa olioissa olevat viittaukset päivitetään osoittamaan viitattujen olioiden lopulliseen paikkaan keon alussa. Lohkokopiointiongelmaa voi ratkaista myös muilla tavoilla.

7. Roskienkeruun tutkimusvälineet

Roskienkeruututkimuksessa ehdottomasti käytetyimmät välineet ovat olleet 2000-luvun alusta lähtien IBM:n avoimen lähdekoodin Jikes RVM Java-virtuaalikone (ks. [Alpern *et al.*, 2000]) yhdessä SPEC-yhtymän JVM98 benchmark -testien kanssa [1999]. Jikes-virtuaalikoneen keskeisin alijärjestelmä, MMTk (*Memory Management ToolKit*), on komponenttipohjainen laajennettava kehys, joka tarjoaa joustavia välineitä roskienkeruun toteutukseen.

Yhteinen alusta ja yhteiset mittarit ovat tehneet tutkimustuloksista vertailukelpoisempia ja tutkimusmetodeista toistettavampia. Jikes ja SPECjvm98 -parivaljakon käyttö tutkimuksessa tuntuu olevan de facto -vaatimus.

7.1. SPEC

Kuten edellä todettiin, tuoreessa kirjallisuudessa eri menetelmien suoriutumisvertailut perustuvat lähinnä SPEC -yhtymän (*System Performance Evaluation Corporation*) JVM98 benchmark -testien käyttöön. SPECjvm98 on kahdeksan Java-ohjelman benchmark-sarja (*_200_check*, *_201_compress*, *_202_jess*, *_209_db*, *_213_javac*, *_222_mpegaudio*, *_227_mtrt* ja *_228_jack*), joista viisi ovat "oikeita" sovelluksia tai niiden johdannaisia. Kaikki testit mittaavat Java-virtuaalikoneen tehokkuutta, käytännössä JIT-kääntäjän, ajoaikaisen järjestelmän, käyttöjärjestelmän ja laitealustan yhteistehoa. Seitsemän testeistä on tarkoitettu suorityskykyarvojen tuottoon ja yksi virtuaalikoneen toimintojen (ja siten muiden testien tulosten) validointiin (*_200_check*).

Sarjan testiohjelmien valinnassa on huomioitu monenlaisia seikkoja, kriteereinä olivat muiden muasssa

- kattava ja laaja tavukoodisisältö,
- litteä ajoprofiili (ei viivytä pienissä ohjelmasilmukoissa),
- toistettavuus,
- muistinkäyttö ja allokaatiotiheys sekä
- käsky- ja datavälimuistiosumien ja hutien tiheys.

Testiohjelmat tekevät sekä kokonaisluku- että liukulukulaskentaa, suorittavat kirjastokutsuja ja IO-operaatioita. Sen sijaan ikkunointi-, verkkoyhteys- ja grafiikkatehtävät on jätetty tarkoituksella pois. Dieckmann ja Hölzle [1999] laativat tarkan selonteon testien toiminnasta ja soveltuvuudesta mittaamaan roskienkeruun erilaisia aspekteja. He kuvaavat matriisimuodossa muiden muassa testiohjelmien olioiden koko-, ikä- ja tyyppijakaumia sekä listaavat muita relevantteja tunnuslukuja, joita on kirjallisuudessa käytetty testien profilointiin.

7.2. Jikes RVM ja MMTk

Vuonna 1997 IBM käynnisti Jalapeño-projektin päämääränään virtuaalikonetekniikoiden tutkimukseen ja testaukseen soveltuvan alustan kehittäminen. MMTk-työkalupakin syntyyn vaikutti Massachusettsin yliopistossa kehitetty GCTk (Garbage Collection Toolkit) -lisäohjelma, jolla korvattiin Jalapeño-virtuaalikoneen omia roskienkeruumekanismeja. IBM teki yhteistyötä kahden yliopiston kanssa kehittääksen GCTk-työkalusetin tilalle uutta ratkaisua. Uuden järjestelmän piti olla joustava kuin GCTk, mutta piti samalla suoriutua vähintään yhtä hyvin kuin olemassaolevat hienosäädetyt kerääjät. MMTk-kirjastossa modulaarisuus ja tehokkuus yhdistyvät suunnittelumallien ja agressiivisen kooperatiivisen kääntämisen kautta. Kirjaston avulla Jikes-virtuaalikone on helposti muutettavissa (erityisesti roskienkeruun osalta), joten suurin osa Jikes-käyttäjistä ovat uusia menetelmiä kehittäviä tutkijoita.

MMTk on tehty Java-kielellä Java-virtuaalikoneita varten, mutta kirjastoa on siirretty menestyksekkäästi muille alustoille ja jopa muihin ohjelmointikieliin. Siirrettävyyteen on päästy käyttämällä kapeaa rajapintaa suoritusjärjestelmän ja muistinhallinnan välillä. Tehoa järjestelmään on saatu ennenaikaisen ("puolistaattisen") kääntäjän (*ahead-of-time compiler*) avulla, joka voi toimia myös JIT-kääntäjänä. Jokaisen viittaustallennuksen yhteyteen kääntäjä laventaa määriteltyä kirjoitusmuurikoodia.

Toki ohjelmointialustan suoritusjärjestelmän toteutus alustalla itsellään asettaa tiettyjä haasteita. Muistin suoraa käsittelyä varten MMTk sisältää jonkin verran natiivia ohjelmakoodia. Javan tyyppijärjestelmää on laajennettu sisältämään muistiosoitteita ja kieleen on lisätty operaatioita näiden käsittelyyn. Aivan erityinen haaste on kerääjän itsensä käyttämän keon turvallinen siivous, sillä kirjaston koodi ja ajoaikainen tila sijaitsevat siivottavassa muistissa (johtuen Jikes-virtuaalikoneen *Java-in-Java* -toteutuksesta).

MMTk-kirjasto toimii määriteltyjen toimintatapojen (*policy*) kautta. Toimintatapa liittää määrättyyn keon osioon (*space*) allokaatio- ja

keruumekanismin. Yksittäinen roskakerääjä on yleensä useamman toimintatavan, osion ja mekanismin kompositio.

MMTk tukee seuraavia allokaatiomekanismeja:

- *Bump-Pointer* -varaaja käyttää keko-osoitintekniikkaa ja
- *Free-List* -varaaja vapaalistaa.

Toimintatapoja voidaan luoda seuraavien mekanismien avulla:

- *Copy space* -osioihin sovelletaan keko-osoitintekniikkaa ja jäljittävää kerääjää, joka kopioi elävän datan osiosta pois.
- *MarkSweep space* -osioissa käytetään vapaalistaa ja jäljittävää kerääjää, joka liittää kuolleet oliot vapaalistaan.
- *RefCount space* -osoissa käytetään vapaalistaa ja viittauslaskentaa.
- *Immortal space* -osiossa käytetään keko-osoitinallokaatiota eikä muistia siivota koskaan.
- *Large object space* -osiossa käytetään karheaa sivuvapaalistaa ja siivoukseen polkumyllytekniikkaa.

Nämä toimintatavat yhdistyvät muodostamaan seuraavat sisäänrakennetut roskakerääjät:

- *SemiSpace*-kerääjä käyttää kahta *Copy space* -osiota stop-and-copy -tyyppistä roskienkeruuta varten.
- *MarkSweep*-kerääjä soveltaa merkitse ja lakaise -tekniikkaa ja laiskaa muistinvarausta yhteen *MarkSweep*-osioon.
- *RefCount*-kerääjä käyttää viivytettyä viittauslaskentaa ja vapaalista-allokaattoria.
- *GenCopy*-kerääjä allokoi klassisen ikäpohjaisen menetelmän tapaan lastentarhana toimivaan *Copy space* -osioon ja ylentää eloonjääneet *SemiSpace*-tyyliin kerätylle alueelle. Kirjoitusmuuri tallentaa viittaukset vanhasta sukupolvesta lastentarhaan. Kun lastentarha tulee täyteen, se siivotaan ja sen kokoa pienennetään eloonjääneiden osuudella. Keko siivotaan kattavasti kun vanhempi osio tulee täyteen.
- *GenMS*-hybridikerääjä toimii kuten *GenCopy*, mutta vanha sukupolvi siivotaan MarkSweep-tyyliin.
- *GenRC*-hybridikerääjä käyttää ulkoista viittauslaskentaa (ks. kohta 9.2) yhdistämään *SemiSpace*-tyyppisen lastentarhan RefCount-tyyppiseen vanhaan sukupolveen.

Käytännössä voidaan "matkia" mitä tahansa kerääjää, joka ei vaadi erityistä laitteistotukea, ja luoda helposti uusia kerääjiä. Tämä mahdollistaa eri menetelmien reilun vertaamisen.

8. Yhdistävyysteoria roskienkeruumenetelmissä

Ikäpohjaisten menetelmien suurin käytännön ongelma lienee sovellusten oliodemografian ennustamattomuus. Ikäpohjaiset menetelmät toimivat tunnetusti hyvin, kun sovellus synnyttää paljon uusia olioita ja suoriutuvat huonosti, kun sovellus käyttää paljon vanhoja olioita. Ikäpohjaisia menetelmiä on paranneltu, mutta ongelman juuret ovat nimenomaan menetelmien perusolettamuksessa. Yhdistävyyspohjainen roskienkeruu on saanut alkunsa kiinnostuksesta vakaammin toimivia osiointimenetelmiä kohtaan.

Hirzel ja kumppanit [2003] esittelivät tehokkaan osittaisen menetelmän, joka perustuu heidän omiin aikaisempiin [2002] tutkimuksiinsa siitä, että eri olioiden väliset viittausyhteydet ja olioiden elinkaaret liittyvät yhteen. Hieman yksinkertaistaen voisi sanoa, että viittauksin yhdistetyt oliot kuolevat usein yhdessä. Menetelmänä yhdistävyyspohjainen roskienkeruu (*connectivity-based garbage collection, CBGC*) on uusi lisäys algoritmien taksonomiaan.

8.1. Harris-analyysi

Yhdistävyyttä on hyödynnetty ennenkin roskienkeruussa. Harris [1999] kehitti analyysimenetelmän, joka hyödyntää käännösaikaista tietoa luokkien (tyyppien) välisistä viittaussuhteista, hieman samaan tapaan kuin olioverkkoa läpikäyvät algoritmit tekevät keräysvaiheessa. Harrisin tarkoituksena oli luoda menetelmä, joka mahdollistaa muistin tehokkaan kierrättämisen varhaisen lakaisun avulla. Harris sovelsi menetelmänsä Bakerin [1992] polkumylly-algoritmiin ja onnistui pienentämään sen jalanjälkeä.

Harris-analyysi on kaksivaiheinen toimenpide. Analyysin ensimmäisessä vaiheessa kuvataan luokkien väliset riippuvuussuhteet asyklisen suunnatun verkon (*dag*, *directed acyclic graph*) muodossa. Luokkien välissä on kaksi relevanttia riippuvuussuhdetta:

- jälkeläissuhde, merkitään *jälkeläinen* < *vanhempi* (<* merkitsee transitiivista relaatiota), ja
- suora viittaus (viittausmuuttuja datajäsenenä), merkitään *viittaajaluokka* → *viitattu luokka*.

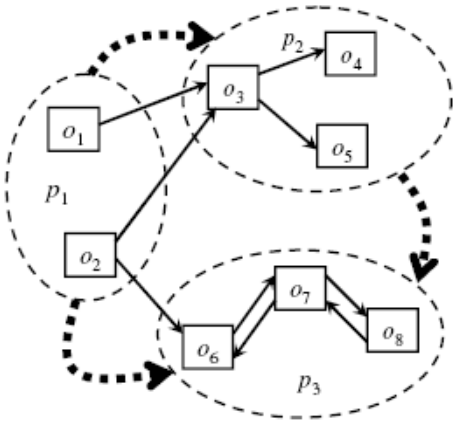
Mikäli kahden luokan välillä on olemassa tällainen riippuvuussuhde, se kuvataan ”voi viitata” –riippuvuudella, merkitään $luokka_1 \Rightarrow luokka_2$ ($\Rightarrow^*$ merkitsee transitiivista relaatiota). Täten

$c_1 \Rightarrow c_2$ jos

- $c_1 \rightarrow c_2$ tai
- $c_1 <^* c_3 \wedge c_3 \Rightarrow c_2$ tai
- $c_1 \Rightarrow c_3 \wedge c_2 <^* c_3$.

Analyysin toisessa vaiheessa luokkagraafi supistetaan osiograafiksi siten, että sen vahvasti kytketyt komponentit (*SCC*, *Strongly Connected Component*)

suljetaan osioiden sisään. Vahvasti kytketty komponentti tarkoittaa tässä sellaista luokkajoukkoa, jossa on maksimaalinen määrä sellaisia luokkapareja, joihon pätee $c_1 \Rightarrow^* c_2 \wedge c_2 \Rightarrow^* c_1$. Osiot muodostavat sellaisen transitiivisen sulkeuman P, että jos $p_i < p_j$ (p_i, $p_j \in$ P), niin osiosta p_i voidaan viitata osioon p_j, mutta ei toisin päin, ei suoraan eikä välillisesti.



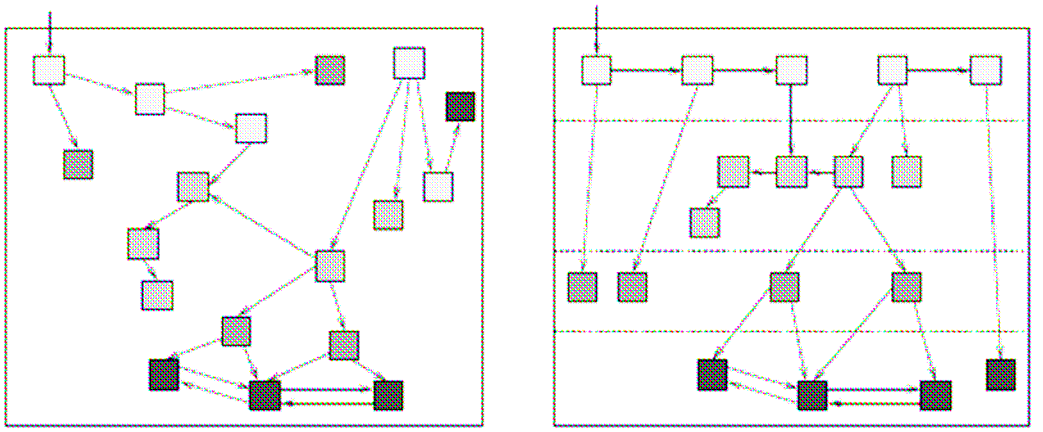
Kuva 5 Osioitu olioverkko. Neliöt ovat olioita, nuolet viittauksia, soikiot osioita ja katkoviivanuolet niiden välillä ovat osiokaaria [Hirzel *et al.*, 2003]

Kuvasta 6 voidaan nähdä globaalin olioverkon (*global object graph, GOG*) osioimattomana ja osioituna. Osioidussa keossa viittaukset osoittavat joko saman osion sisään tai alempana olevaan osioon. Roskienkeruu lähtee sulkeuman juuresta ja etenee osioittain. Kerääjä muistaa kaikki kerättävästä osiosta lähtevät viittaukset, joten osio voidaan lakaista heti kun kerääjä on siirtynyt seuraavaan osioon. Algoritmin kulkua käsitellään tarkemmin kohdassa 8.3.

Staattisissa kielissä kaikki luokat ja niiden väliset suhteet tunnetaan ohjelman käynnistyessä, sen sijaan esimerkiksi Java-kielessä luokat ladataan tarpeen mukaan ja siten usein joudutaan päivittämään yhdistävyysgraafia ainakin ohjelman alkuvaiheessa tiheään. Toisaalta graafi pysyy tiiviinä ohjelman koko suorituksen ajan. Harrisin analyysiä on sovellettu laajalti muun muassa luokkalataus- ja JIT-toteutuksissa, lisäksi se toimii CBGC:n osiointifunktion referenssitoteutuksena.

Analyysimenetelmän hyödyllisyyttä vähentävät tyypiltään liian yleiset suorat viittaukset luokkien välillä. Ongelmallisia ovat esimerkiksi Java-kielen parametroimattomat kokoelmaluokat, esimerkkinä silpputietorakennetta edustava luokka `java.util.Hashtable`, jossa sekä avaimina, että arvoina käytetään luokan `java.lang.Object` olioita. Käytännössä ei tarvita tämänlaista yleisyyttä missään sovelluksessa, vaan avaimen ja arvon tyyppi tarkistetaan ja muunnetaan ajoaikana spesifiseksi. Analyysifunktio ei kuitenkaan ole spesifisistä tyypeistä tietoinen ja luokkahierarkiassa korkealla

sijaitsevia luokkia sisältävät osiot saattavat paisua suuriksi. Java-kielen parametroidut kokoelmatyypit ovat ratkaisu tähän ongelmaan, kunhan ohjelmoija hyödyntää niitä hyvien käytäntöjen mukaisesti.



Kuva 6 Osioimaton ja Harris-analyysillä osioitu keko. [Harris, 1999]

8.2. Tilastoja olioverkon yhdistävyydestä

Hayes [1991] huomautti, että ohjelmassa avainasemassa olevat oliot vievät kuollessaan mukanaan koko joukon muita oliota ja synnyttävät samalla mahdollisuuden kerätä roskia opportunistisesti (*key object opportunism*). Opportunismi on tällöin sekä ajallista että paikallista, sillä sekä aika että paikka ovat tiedossa tehokasta muistinsiivousta varten. Hirzel ja kumppanit [2002] tekivät konkreettisia mittauksia olioiden välisten yhdyssiteiden ja kuolinajankohdan korrelaatiosta jäljittämällä 22 ohjelmaa. He jakoivat oliot elinkaarensa pituuden perusteella neljään kategoriaan, jotka ovat:

- aidosti kuolemattomat (*truly immortal*),
- lähes kuolemattomat (*quasy immortal*),
- lyhytikäiset (*shortlived*) ja
- pitkäikäiset (*longlived*)

oliot. Kategorisoinnin kriteerit ovat samassa järjestyksessä

- *kuolinhetki = ohjelman loppu*,
- *elinaika > ohjelman loppu – kuolinhetki*,
- *elinaika < kynnys* ja
- kaikki muut.

He laskivat, että kaiken kaikkiaan 58,5% olioista olivat lyhytikäisiä, 2.3% pitkäikäisiä, 0.5% lähes kuolemattomia ja 35,4% aidosti kuolemattomia. Ainoastaan pinosta viitattuja olioita oli 36,1%, joista 97% olivat lyhytikäisiä. Olioista 17,3% oli globaaleista muuttujista transitiivisesti viitattuja, niistä 65% olivat aidosti kuolemattomia.

He laskivat myös todennäköisyydet kahden olion kuoleman samanaikaisuudelle kategorioittain, tulokset on esitetty taulukossa 1. Yhdistävyyden he kuvasivat funktion *Connectivity* avulla:

Connectivity(O_1, O_2), O_1, $O_2 \in$ GOG.

Tilastointia varten he määrittelivät viisi (osittain päällekkäistä) luokkaa:

- *pointsTo*(O_1, O_2),
- *pointsTo*(O_1, O_2) $\wedge$ *mutated*(O_1),
- *pointsTo*(O_1, O_2) $\wedge \neg$*mutated*(O_1),
- *SCC*(O_1) = *SCC*(O_2) ja
- *WCC*(O_1) = *WCC*(O_2),

jotka ovat selkokielellä:

- olio O_1 viittaa olioon O_2,
- olio O_1 viittaa olioon O_2, mutta viittaus poistetaan,
- olio O_1 viittaa olioon O_2 eikä viittausta poisteta,
- oliot O_1 ja O_2 ovat vahvasti kytkettyjä (viittauspolku on olemassa molempiin suuntiin) ja
- oliot O_1 ja O_2 ovat heikosti kytkettyjä (viittauspolku on yksisuuntainen).

Todennäköisyys sille, että O_1 ja O_2 kuolevat yhdessä (i), saadaan jakamalla samaan aikaan kuolevien olioparien (*equideath pair*) lukumäärän kaikkien parien lukumäärän kanssa. Kohta (i) voidaan esittää seuraavasti:

i) Probability{$t_{death}(O_1) = t_{death}(O_2)$}.

***Connectivity*(O_1, O_2)**	***Probability*{$t_{death}(O_1) = t_{death}(O_2)$}**
O_1, O_2	33,1%
pointsTo(O_1, O_2)	76,4%
pointsTo(O_1, O_2) $\wedge$ *mutated*(O_1)	61,1%
pointsTo(O_1, O_2) $\wedge \neg$*mutated*(O_1)	83,4%
SCC(O_1) = SCC(O_2)	72,4%
WCC(O_1) = WCC(O_2)	46,3%

Taulukko 1 Kuolevatko yhdistetyt oliot yhdessä? [Hirzel *et al.*, 2002]

Edellisten lisäksi Hirzel ja kumppanit laskivat, kuinka monen muun olion ulottuvissa oliot keskimääräisesti olivat. Tämä tunnusluku toimii karkeana arviona siitä, kuinka helposti olio on kerättävissä. He saivat tulokseksi lyhytikäisille olioille 1 – 2 ja aidosti kuolemattomille 40 – 80.

Nämä tilastotiedot johtivat päätelmiin siitä, että

- viittausjuuren tyyppi korreloi viitatun olion eliniän kanssa,

- yhdistetyt oliot on syytä roskakerätä yhdessä ja
- yhdistävyystiedon perusteella lyhytikäiset oliot ovat löydettävissä ja helposti kerättävissä.

8.3. CBGC-osiointi

Osittaisena roskienkeruumenetelmänä CBGC osioi keon ja kerää roskat vain tietyistä osioista kerralla. Menetelmä luo osiot suorittamalla yhdistävyys-analyysin ohjelmasta käyttämällä analyysifunktiona esimerkiksi Harris-analyysiä. Yhdistävyysanalyysin tuloksena on suunnattu asyklinen verkko erillisiä osioita. Osiot ovat itsekin suunnattuja verkkoja, joiden yhdistävyys on mahdollisimman hyvä, eli verkon solmut ovat mahdollisimman vahvasti kytkettyjä komponentteja. Kaaret edustavat potentiaalisia viittauksia (*may-point-to relationship*) komponentista (luokasta) toiseen. Osioiden rajoja ylittävät viittaukset kuitataan osiokaarin, eli kaarella osioiden välillä. Analyysi on konservatiinen, mikä tarkoittaa, että jos kahden luokan välillä viittaukset ovat mahdollisia, niin luokat ovat joko samassa osiossa, tai osioita yhdistää kaari. Kaaren olemassaolo ei kuitenkaan ole tae siitä, että kahden osion välillä todellakin on mahdollinen riippuvuussuhde.

Kun kerääjä on valinnut sopivan osion, se ottaa keruun kohteeksi kaikki sen edeltäjät hyppimällä osiograafissa vastavirtaan ja edeten alusta kohti kohdeosiota. Tällöin keräämättömistä osioista ei siis voida viitata juuri kerättävän alueen sisään, kirjoitusmuuria ei siksi tarvita eikä nepotistisia tilanteita pääse syntymään. Kerättävä osio on lakaistavissa heti kun roskienkeruu on siinä päättynyt. Monisuoritinjärjestelmissä riippumattomat osiot voidaan kerätä samaan aikaan pienellä synkronointityöllä.

Menetelmä mahdollistaa hyviä vasteaikoja, koska koko kekoa ei tarvitse koskaan kerätä samalla kertaa, muuten kuin siinä patologisessa tilanteessa, jossa yksi osio on riippuvainen kaikista muista osioista. Paikallisuusongelmaan menetelmä tarttuu varaamalla samaan osioon kuuluville olioille yhteisen allokaatioalueen, sillä muutin käyttää suurella todennäköisyydellä näitä olioita yhdessä.

8.4. CBGC-keruu

Keräykset suoritetaan opportunistisesti valitsemalla keräyskohde kustannus-hyöty -analyysin perusteella. Analyysi tehdään kahden funktion, arvioijan (*estimator*) ja valitsijan (*chooser*), avulla. Arvioija päättelee elävien ja kuolleiden olioiden suhdetta jokaisessa osiossa ja olioiden mätänemisnopeutta (*rate of decay*). Valitsijafunktion tehtävänä on valita transitiivisesti suljetun osiojoukon arvioijan tulosten perusteella siten, että kerääjä voi vapauttaa mahdollisimman paljon muistia mahdollisimman pienin kustannuksin.

Funktioiden toteutukseksi Hirzel ja kumppanit [2003] esittelevät joitakin ratkaisuja: Yhdistetty arvioijafunktio perustaa alustavan arvionsa osion mätänemisnopeudesta havaintoon viittausjuuren tyypin (paikallismuuttuja, staattinen muuttuja) vaikutuksesta viitattujen olioiden elinikään. Ensimmäisen roskienkeruun jälkeen mätänemistä ennustetaan radioaktiivisen hajoamismallin mukaan: Olkoon osion *p* mätänemisfaktori *d*, keskivertoikä *a*, olioiden lukumäärä *n*, olioiden selviämisfaktori *S* sekä edellisestä allokoinnista kulunut aika *t*. Kun osioon *p* allokoidaan uusi olio, keskivertoikä *a* päivitetään (i) ja jokaisen roskienkeruun jälkeen osion *p* mätänemisfaktori lasketaan uudestaan (ii). Kohdat (i) - (ii) voidaan esittää seuraavasti:

i) $a \leftarrow (a + t) \cdot (n / (n + 1))$

ii) $d \leftarrow -\ln(S) / a$.

Valitsijafunktion tehtävää voidaan formuloida näin: Etsi sellainen suljettu osajoukko *C* kaikkien osioiden joukosta *P* ($C \subseteq P$), joka funktion *quality* (iii) parametrina maksimoi funktion paluuarvon:

iii) $quality(C) = \sum_{p \in C} dead(p) / \sum_{p \in C} live(p)$; $dead(q), live(q) \rightarrow \{0, 1\}$, $q \in P$.

Funktio *quality* ilmaistaan siis hyödyn (vapautettavan muistin) ja kustannusten (elävien olioiden läpikäynti, kopiointi, yms.) osamääränä.

Käytännössä osiointi on ratkaistu siten, että osiot koostuvat lohkoista, joiden koko on jokin kahden potenssi (esimerkiksi 2^{10}, 1024 tavua). Uudet oliot varataan osion viimeiseen lohkoon. Minkä tahansa olion osio on pääteltävissä sen osoitteesta. Roskienkeruuseen käytetään Cheneyn [1970] *breadth-first* -algoritmia. Kohdeavaruudeksi varataan kaikille osioille tarvittaessa omat uudet lohkot.

8.5. Tehokkuus vertailussa

Mitataakseen CBGC:n potentiaalia tekijät laativat optimaalisia (samalla epärealistisia) funktioita simulaatioita varten. Esimerkiksi lähes täydellisesti toimivan arvioijafunktion he saivat aikaan jäljittämällä testiohjelmat keräten niistä tarkat tiedot olioiden syntymä- ja kuolinajoista valmiiksi. He vertasivat CBGC:n esimerkki- ja optimaalitoteusta Appel-kerääjään. Tavallinen CBGC:n suoritus oli Appelin veroinen, mutta optimaalinen CBGC suoriutui lähes aina parhaiten. Erilaisten mittareiden perusteella voidaan hyvin olettaa, että kehittelemällä CBGC-osatoteutuksia (osiointi-, arvioija- ja valitsijafunktiot) voidaan päästä ikäpohjaisia ratkaisuja parempiin tuloksiin.

9. Muita uusia tutkimusaiheita ja menetelmiä

Tässä luvussa katsastetaan lyhyesti joitakin 2000-luvun saavutuksia ja tutkimusaiheita, jotka koskevat olioiden kuolinhetken ennustamista, ajoaikaisia algoritminvaihtoja, kahta merkittävintä uuttaa ikäpohjaista kerääjää (*Older First* ja *Beltway*) sekä roskienkeruun ja käyttöjärjestelmän virtuaalimuistimekanismien välistä tehostettua yhteistyötä.

9.1. Pinoperustainen osiointi ja pakoanalyysi

Kerättävien olioiden ryhmittäminen tai keon osiointi voidaan tehdä ainakin olioiden iän, viittaussuhteiden ja syntymäpaikan (*allocation site*) perusteella. Syntymäpaikka, eli pinokehys (*stack frame*), josta käsin olio luodaan, on pinopohjaisten ositusmenetelmien käyttämä peruste. Menetelmät perustuvat oletukseen siitä, että olioiden elinikä liittyy niitä luoneen pinokehyksen elinikään ja yrittävät opportunistisesti vapauttaa kehyksessä luodut oliot tämän poiston yhteydessä.

Park ja Goldberg [1992] tutkivat mahdollisuutta luoda osan olioista suoraan pinon jatkeeksi käyttäen viittausten pakoanalyysiä (*escape analysis*). Pakoanalyysissä tutkitaan, syntyykö olioon ulkopuolisia viittauksia, esimerkiksi olion syntymämetodin, -säikeen, -lohkon yms. tutkittavan alueen ulkopuolelta. Alueellista muistinhallintaa (*region-based memory management*) ja pakoanalyysiä käyttävät Qian ja Hendren [2002] menetelmässään, jossa pinokehyksiin liitetään allokaatioalue kehyksessä luotuja olioita varten. Jos alueelta pakenee olioita, alue liitetään globaaliin roskakerättyyn alueeseen, muuten koko alue vapautetaan kertaheitolla, kun kehys ponnautetaan pinosta. Cannarozzi ja kumppanit [2000] käyttävät menetelmässään dynaamisia tarkistuksia päätellessään, mikä on alin pinokehys, josta johonkin olioon on viitattu. Olion tarpeellisuutta ei tarvitse tutkia ennen kuin kyseinen kehys on ponnautettu.

Aikaisemmin ajateltiin, että nämä menetelmät soveltuvat lähinnä sovelluksiin, jotka on kirjoitettu jollakin funktionaalisella ohjelmointikielellä, koska tällaiset kielet suosivat luonnostaan alueystävällistä ohjelmointityyliä. Choi ja kumppanit [1999] kuitenkin sovelsivat pakoanalyysiä Javaan lisäämällä tukea poikkeuksille, säikeille ja try/catch–lohkoille. Pakoanalyysin avulla he onnistuivat eliminoimaan merkittävän osan lukitustilanteista ja toteuttivat pino-allokaatiomekanismin, joka vähentää synkronointi- ja roskienkeruutarpeita.

9.2. Adaptiiviset kerääjät

Useimmissa käytännön roskienkeruutoteutuksissa ei toimita vain yhden algoritmin varassa. Suoritusjärjestelmä voi valita uuden algoritmin spekulatiivisesti heurististen sääntöjen avulla. Esimerkiksi muistin

pirstoutuessa mark-and-sweep -algoritmi vaihtuu kätevästi mark-compact -algoritmiin. Joissakin ohjelmissa kopioiva algoritmi voi olla tehokas ohjelman alustusvaiheessa, jossa syntyy paljon uusia olioita, mutta ei enää myöhemmin, kun ohjelman muistinkäyttö on vakiintunut ja vanhoja olioita kopioidaan jatkuvasti paikasta toiseen. Tällöin on tarkoituksenmukaisempaa vaihtaa mark-and-sweep -tyyppiseen algoritmiin. Eckel [2000] toteaa, että Sun Microsystemsin JDK:ssa 1.1 virtuaalikone toimii juuri edellä mainitulla tavalla. Bacon ja kumppanit [2003] esittivät C++-kieleen tarkoitetun toteutuksen, joka käyttää tavallisesti ei-kopioivaa algoritmia, mutta vaihtaa automaattisesti kopioivaan algoritmiin muistin pirstoutuessa. Tämäntapaiset heuristiset algoritmivaihdot ja hybridiratkaisut ovat yleisiä eri menetelmissä.

Kirjallisuudestakin löytyy kokeellisia ja analyyttisiä tuloksia, jotka osoittavat, kuinka tietyt algoritmit soveltuvat paremmin joihinkin tilanteisiin kuin muut ja kuinka algoritminvaihto tietyissä tilanteissa voi olla kannattavaa. Tutkimustuloksin on osoitettu, että mikään tietty algoritmi ei voi suoriutua parhaiten kaikissa tilanteissa, koska muistinhallinnan tehokkuus on täysin riippuvainen dynaamisista tekijöistä, ohjelman käyttäytymisestä ja resurssien saatavuudesta. Suurin osa menetelmistä kuitenkin pyrkii yleiskäyttöisyyteen jonkin tietyn mekanismin avulla, jota paineen alla mahdollisesti avustaa jokin apumekanismi.

Moni suoritusjärjestelmä, kuten Sun [2005] Microsystemsin Hotspot, tarjoaa useampaa roskienkeruumenetelmää ja optimointimahdollisuutta, mutta määritykset tehdään staattisesti komentoriviltä. Soman ja kumppanit [2003] toteavat, että tällaiset järjestelmät eivät kuitenkaan välttämättä ole riittäviä seuraavan sukupolven palvelinohjelmistoja ajatellen, joissa yksi suoritusjärjestelmä palvelee keskeytyksittä monta dynaamisesti ladattavaa sovellusta ja muuta koodikomponenttia, joiden käyttäytymismallit ja resurssitarpeet voivat olla hyvinkin erilaiset. Tällainen järjestelmä on esimerkiksi NonStop Server for Java -alusta.

Soman ja kumppanit [2003] tutkivat suoritusjärjestelmää, joka voi vaihtaa roskienkeruumenetelmää ajo-aikana dynaamisesti. Järjestelmä pyrkii valitsemaan automaattisesti parhaan mahdollisen allokaatio- ja roskienkeruu-mekanismin vallitsevan muistinkäyttötilanteen ja resurssisaatavuuden (keon koon) mukaan ajoaikaisen profilointijärjestelmän avulla. Tämä virtuaalikone on Jikes RVM –järjestelmän laajennos ja pystyy oletusarvoisesti valitsemaan neljän eri strategiaa edustavan kerääjän välillä ajoaikana.

Somanin ja kumppaneiden kerääjä tarjoaa kolme mekanismia keruumenetelmän valintaan: Ajoaikainen profiloija toimii samaan tapaan kuin Jikes –virtuaalikoneen adaptiivinen optimointijärjestelmä (AOS). Se seuraa

ohjelman allokaatio- ja roskienkeruukäyttäytymistä kunnes jokin kynnysarvo saavutetaan, jolloin heuristisiten sääntöjen avulla päätellään mikä menetelmä olisi ohjelman ja keon koon kannalta tehokkain ja mahdollisesti suoritetaan menetelmänvaihto. Toinen mekanismi on annotaatio-ohjattu menetelmänvalinta, joka perustuu tavukoodiin upotettuihin optimointivihjeisiin. Tiedot vihjeitä varten saadaan ajamalla kyseinen ohjelma useasti erilaisin muistiasetuksin. Järjestelmä valitsee keruumenetelmän vihjeiden perusteella, tosin ajoaikainen profilointi voi lopulta päättyä toiseen ratkaisuun. Kolmas mekanismi on menetelmän valinta ohjelmakoodissa.

Kerääjällä on käytännössä pelkästään akateemista arvoa. Keko on jaettu toteutuksessa osiin siten, että osiot ovat aina tietyssä järjestyksessä ja osa osioista on varattu vain tietyn tyyppisten kerääjien käyttöön. Neljä tuettua kerääjää ovat SemiSpace, MarkSweep, Generational SemiSpace ja hybridi GenMS (ks. kohta 7.2). Ikäpohjaisille kerääjille on varattu yhteinen lastentarha, jota muut menetelmät eivät voi käyttää. Myös MarkSweep- ja SemiSpace-kerääjät käyttävät eri osioita. Näin vaihto voidaan tehdä suorittamalla erikoisroskienkeruu, jossa oliot siirtyvät mahdollisesti osiosta toiseen. Tämä tarkoittaa, että tietty osa kekoa on koko ajan joutilaana odottamassa menetelmänvaihtoa.

Periaatteessa uusia kerääjiä on helppo lisätä, mutta käytännössä uusien kerääjien on ilmeisesti oltava näiden edellä mainittujen variaatioita. Soman ja kumppanit arvelivat, että järjestelmää voisi hyödyntää sovellus- ja ajopalvelimissa. Mainitsemisen arvoinen hybridiratkaisu on myöhempi (tai piilo-) viittauslaskenta (*ulterior reference counting, URC*), jonka esittivät Blackburn ja McKinley [2003]. Menetelmän idea on ikäpohjaisten menetelmien ja viittauslaskennan vahvuuksien, eli hyvän läpäisykyvyn ja hyvien vasteaikojen, yhdistäminen. Menetelmä on ikäpohjainen, mutta soveltaa viittauslaskentaa vanhan sukupolven siivoamiseen.

9.3. Vanhat ensin (Older First)

Sukupolvipohjaiseen roskienkeruuseen on tuonut uutta näkökulmaa hieman erilaista ikäpohjaista osiointia käyttävä Older First, eli ”vanhat ensin” -algoritmi, jonka esittivät Stefanović ja kumppanit [1999]. Algoritmin idea perustuu havaintoon siitä, että vaikka suurin osa olioista kuoleekin nuorena, jokainen olio elää edes vähän aikaan, eli tarvitsee aikaa kuolla. Algoritmi kiertää nuorten olioiden jatkuvasta kopioimisesta syntyvää haittaa organisoimalla keon sisällön olioiden iän mukaan ja aloittamalla roskienkeruun vanhoista olioista. Vanhat ensin -menetelmän idea ei ole täysin uusi, sillä on olemassa ikäpohjaisia menetelmiä, jotka eivät noudata ”nuoret ensin” -periaatetta roskienkeruussa. Esimerkiksi Moonin [1984]

päiväperhokerääjä (*ephemeral collector*) siivoaa vain täyttyneet sukupolvet, iästä riippumatta.

Vanhat ensin menetelmässä uudet oliot varataan allokaatio-osoittimen avulla, joka liukuu keon ”vanhasta” päästä kohti ”nuorta”. Algoritmi ei käytä kiinteätä sukupolvijakoa vaan tietynkokoista ”roskienkeruuikkunaa”, joka seuraa (etäältä) allokaatio-osoitinta tiivistäen ikkunassa eloon jääneet oliot vanhojen olioiden perään. Jos ikkuna saavuttaa allokaatio-osoittimen, se siirtyy takaisin keon vanhaan päähän.

Viittaukset alueelta toiselle joudutaan tallentamaan kirjoitusmuurin avulla, sikäli kun viittausten kohdealue kerätään itsenäisesti lähdealueesta riippumatta. Toisaalta tämä mahdollistaa sen, että kekoa ei tarvitse kerätä koskaan kokonaan (muuten kuin ikkuna kerrallaan). Algoritmin vahvuus on siis siinä, että uusilla olioilla on riittävästi aikaa kuolla ennen kuin roskienkeruuikkuna saavuttaa ne, lisäksi keruu rajoittuu pienelle alueelle kerrallaan, mikä pienentää paikallisuusongelmaa roskienkeruun aikana.

Stefanović ja kumppanit [2002] tekivät Jikes RVM -toteutuksen vanhat ensin –algoritmista verratakseen SPEC-sarjojen [1999, 2001] avulla sen tehoa ikäpohjaisiin menetelmiin, perinteiseen menetelmään ja vaihtuvanmittaista sukupolvikokoa käyttävään Appelin [1989] kerääjään. He laskivat kuinka ikäpohjaisten menetelmien kahdentyyppiset kustannukset voidaan tasapainottaa. Kyseiset kustannukset syntyvät olioiden kopioinnista ja viittausten paikannuksesta, esimerkiksi kirjoitus- ja lukumuureista sekä kopio-osoittimista. Perinteiset ikäpohjaiset menetelmät suorittavat enemmän kopiointia ja selvästi vähemmän viittauspaikannusta kuin vanhat ensin -algoritmi. Jälkimmäinen on kuitenkin suoriutunut kaikista testeistä perinteistä ikäpohjaista algoritmia paremmin ja monessa testissä Appelin algoritmia paremmin, vaikka sitä selvästi hidastaa vanhojen olioiden ”turha” kopiointi.

Vanhat ensin -menetelmä ei näytä ottavan kantaa kelluvaan roskaan. Ajatellaan esimerkiksi konservatiivista kirjoitusmuuria, joka tallentaa kaikki roskienkeruuikkunan eteen osoittavat viittaukset. Ikkuna palaa keon vanhaan päähän suhteellisen harvoin, eli paljon olioita ehtii kuolla ennen kuin näin tapahtuu. Tämä puolestaan johtaa nepotistiseen järjestelmään, koska kirjoitusmuurin tallentama osoitinjoukko ei pysy ajan tasalla ja siten pitää olioita perusteettomasti hengissä, sillä osa viittaavista olioista on jo ehtinyt kuolla ja osa viittauksista on ehtinyt päivittyä. Tekijät eivät totea, että lyhyiden taukojen ja lyhyen keruun kokonaiskeston hintana on kelluvan roskan paljous ja paisunut ohjelman jalanjälki, eivätkä mittaa näitä algoritmin aspekteja tai vertaa muiden tutkittavien algoritmien vastaaviin arvoihin.

9.4. Beltway-kerääjä

Roskienkeruututkimuksessa on vuosikymmenten mittaan tehty viisi menetelmien kehityksen kannalta olennaista havaintoa. Blackburn ja kumppanit [2002] toteuttivat ensimmäisen roskienkeruujärjestelmän, joka rakentuu kaikkien näiden havaintojen varaan. Heidän Beltway-kerääjänsä on ikäpohjaisia kopioivia algoritmeja yleistävä kehys.

Edellisen kappaleen kyseiset viisi havaintoa ovat jo kaikki tulleet vastaan edellisissä luvuissa. Ensimmäinen niistä on Ungarin [1984] heikko sukupolvihypoteesi, jonka mukaan oliot kuolevat nuorina. Täten roskienkeruu kannattaa järjestää nuorimpien olioiden joukossa. Toinen havainto seuraa edellisestä suoraan: vanhojen olioiden joukossa on vähän roskia. Kolmannen havainnon tekivät Stefanović ja kumppanit [1999], jotka huomauttivat, että on syytä antaa olioille aikaa kuolla (ks. kohta 9.3). Neljäs havainto koskee roskienkeruun vähittäisyyttä. Siedettäviin vasteaikoihin pyritään pitämällä lastentarha pienenä ja keräämällä siitä roskat melko lyhyin väliajoin. Perinteiset ikäpohjaiset menetelmät vaativat kuitenkin sen, että aina välillä järjestetään koko kekoa koskeva roskienkeruu, siksi nämä menetelmät eivät anna takeita keruun enimmäiskeston suhteen. Ongelman ratkaisivat muiden muassa Hudson ja Moss [1992] sekä Stefanović ja kumppanit [1999] aggressiivisemman vähittäisyyden avulla, siivoamalla muistia (enintään yksi) osio kerrallaan. Viimeinen havainto liittyy kopioivien menetelmien viittauspaikallisuutta kohentavaan vaikutukseen. Hayes [1991] huomautti, että muutin käsittelee usein samanikäiset oliot yhdessä. Kopioivilla algoritmeilla on mahdollisuus järjestää samanikäiset oliot keon samaan osaan parantaen väli- ja virtuaalimuistitehokkuutta (sivutusta, välimuisti- ja TLB-osumia ajatellen).

Beltway-kerääjä organisoi keon kaksitasoiseksi vyöhykkeiden ja osioiden avulla. Osio (*increment*) on itsenäisesti kerättävä muistialue. Vyöhyke (*belt*) on kokoelma osioita, joissa roskienkeruu suoritetaan FIFO-periaatteella (*first-in-firts-out*). Kerääjän toimintaperiaate mahdollistaa kaikkien yllämainittujen havaintojen huomioon ottamista ja hyödyntämistä. Sukupolvia edustavat vyöhykkeet, alin vyöhyke edustaa lastentarhaa. Keräämällä roskat alimmasta vyöhykkeestä keskitytään nuoriin olioihin ja vältetään vanhojen olioiden läpikäyntiä. Mielivaltaiseen vähittäisyyteen päästään suorittamalla keruu osio eikä sukupolvi kerrallaan, eli vähittäisyys ei ole enää kytköksissä sukupolven kokoon. Koska vyöhykkeet on jaettu FIFO-järjestyksessä kerättäviin osioihin, keruu aloitetaan aina vyöhykkeen vanhimmista olioista, näin ollen olioilla on aikaa kuolla. Ikäpohjainen osioiva ja kopioiva algoritmi on tae hyvästä viittauspaikallisuudesta.

Beltway-kerääjä voidaan konfiguroida toimimaan kuten mikä tahansa tunnettu kopioiva tai ikäpohjainen menetelmä valitsemalla oikeat osiokoot, vyöhykeasettelut ja ylennysstrategiat. Yksinkertaisin esimerkki lienee SemiSpace-kokoonpano, joka koostuu yhdestä kahteen samankokoiseen osioon jaetusta vyöhykkeestä, mutta myös Appel ja Older First -kerääjät voidaan matkia yhtä helposti. Kahden edellä mainitun kerääjän etuja yhdistää Beltway X.X -niminen kokoonpano, jossa X ilmaisee osioiden prosentuaalista kokoa ja samalla vähittäisyyden astetta. Keko on jaettu kahden vyöhykkeen kesken, mutta alempi vyöhyke (lastentarha) voi kasvaa ylemmän kustannuksella tiettyyn rajaan asti. Käytettävän muistin loppuessa siivotaan lastentarhan vanhin osio kopioiden eloonjääneet ylemmän vyöhykkeen nuorimpaan osioon. Kun lastentarha tyhjenee, loppuu samalla käytettävä muisti ylemmästä vyöhykkeestä. Vyöhyke siivotaan alkaen vanhimmasta osiosta ja eloonjääneet siirretään nuorimpaan osioon. Koko kekoa kattavaa siivousta ei tehdä, eli Beltway X.X -kokoonpano ei takaa, että kaikkia roskia siivotaan. Jos halutaan päästä kelluvasta roskasta täytyy käyttää Beltway X.X.100 -kokoonpanoa, joka luopuu osittain vähittäisyydestä kattavuuden hyväksi.

Beltway-kerääjän teknisen toteutuksen tehokkuus on taattu tehokkaiden kirjoitusmuurien, keruuliipaisinten ja dynaamisesti mitoitettujen kopiointivarojen avulla. Kirjoitusmuurien toimintaa on helpotettu toteuttamalla osiot muistikehyksinä, yhtenäisinä muistialueina, joiden alkuosoite on jokin kahden potenssi. Kirjoitusmuurissa osionsisäiset ja -ulkoiset viittaukset voidaan erottaa toisistaan bittien sivuttaissiirtokäskyn ja vertaamisoperaation avulla. Kehyksillä on myös järjestysnumero, joka ilmoittaa kehyksen vuoron roskienkeruussa. Kirjoitusmuurin ei tarvitse tallentaa niitä osioiden välisiä viittauksia, joiden lähdeosion vuoro on kohdeosiota aikaisempi.

Aina ei ole käytännöllistä aloittaa roskienkeruuta vasta, kun keko on täynnä. Beltway-kerääjän käytössä on kolme liipaisinta, eli mekanismia, jonka avulla roskienkeruu voidaan ajoittaa: Lastentarha-liipaisin edustaa perinteistä käytäntöä aloittaa roskienkeruu kun lastentarha tulee täyteen. Remset-liipaisin perustuu tallennettujen osoittimien lukumäärän seuraamiseen. Mitä enemmän toissijaisia juuria kertyy, sitä enemmän olioita jää roskienkeruusta eloon ja sitä kauemmin niiden läpikäynti kestää. Siksi lukumäärän ylittäessä tietyn kynnyksen kannattaa aloittaa roskienkeruu. Time-to-die -liipaisimen avulla voidaan asettaa olioille vähimmäisikä (*TTD, time-to-die*). Aika mitataan allokoituina muistitavuina. Kun kekoon jää enää ennalta määritelty määrä (TTD) vapaata muistia, uudet oliot allokoidaan uuteen osioon. Tämä takaa sen, että vanhassa osiossa olioiden ikä on suurempi kuin TTD.

Perinteiset kopioivat algoritmit tarvitsevat kopiointivaran, joka on yhtä suuri tai vähän suurempi kuin käytettävissä oleva muisti. Beltway-kerääjän tapauksessa kopiointivara voidaan kokoonpanosta riippuen laskea dynaamisesti kerättävien osioiden koon ja käyttöasteen perusteella.

Beltway-kerääjä tarjoaa alustan kopioivien ikäpohjaisten menetelmien vertailuun ja uusien parempien algoritmien kehitykseen. Osoituksena tästä ovat suoritukseltaan muiden muassa Appel-kerääjää ylittävät Beltway X.X ja Beltway X.X.100 -kokoonpanot.

9.5. Virtuaalimuisti ja kirjanmerkkikerääjä

Kun fyysinen muisti on vähissä, paikallisuusongelma nousee vahvasti esiin suurten kekojen siivouksessa. Koko kekoa koskettava roskienkeruu saattaa aiheuttaa paljon sivutusta ja siten pitkiä taukoja, jos olioverkko on hajaantunut muistissa. Moni algoritmi kiinnittää huomiota paikallisuusongelmaan, mutta Hertz ja kumppanit [2005] tutkivat kokonaisvaltaista ratkaisua. Heidän ratkaisunsa, kirjanmerkkikerääjä (*bookmarking collector, BC)*, tekee yhteistyötä virtuaalimuistin hallintajärjestelmän kanssa ja siten eliminoi roskienkeruusta johtuvaa sivutusta.

Roskienkeruualgoritmit voidaan jakaa kolmeen ryhmään sillä perusteella, kuinka tietoisia ne ovat virtuaalimuistin käytön seurauksista. Osa algoritmeista ei ota kantaa mahdolliseen sivutukseen (*vm-oblivious*), toiset vähentävät sivutustarvetta joko tiivistämällä kekoa pitääkseen ohjelman muistinvaraisena tai vähentämällä tarvetta käsitellä koko kekoa kerralla osioimalla sitä (*vm-sensitive*). Kolmanteen ryhmään kuuluvat tekniikat viestivät virtuaalimuistin hallintajärjestelmän kanssa (*vm-cooperative*). Näin tehostaa sivutusta roskienkeruussa muiden muassa ”päiväperhokerääjä”, jonka Moon [1984] kehitti. Moonin kerääjä vastaanottaa ilmoituksen häädettävästä sivusta ja kirjaa muistiin mihin sukupolviin sivusta viitataan. Kun kerääjä käsittelee jotakin sukupolvea, sen ei tarvitse tutkia kaikkia häädettyjä sivuja.

Moonin kerääjä myös tiivistää keon muistipaineen alla. Vielä tehokkaampi tapa on asettaa keon koko uudestaan vallitsevan virtuaalimuistitilanteen mukaan tiivistämällä kekoa ja hylkäämällä tarpeettomia sivuja. MMTk-työkalusarja tukee tätä toimintoa. Blackburn ja kumppanit [2004] toteavat, että MMTk voi reagoida muutoksiin elävän datan osuudessa ja taukojen pituudessa muuttamalla keon kokoa ja osioiden kokojen suhdetta ennaltamäärätyin tavoin. Myös Sun [2005] Microsystemsin Hotspot-virtuaalikone voi mitoittaa keon uudestaan läpäisykykyä, taukoja ja jalanjälkeä koskevien vaatimusten perusteella, joita voidaan antaa komentoriviparametreina.

Herzin ja kumppaneiden ratkaisu perustuu virtuaalimuistin hallinnointijärjestelmän päätöksenteon ohjaamiseen ja ”tiivistelmän” laatimiseen häädetyistä

sivuista. Kirjanmerkkikerääjä merkitsee häädettävän sivun uumenista viitatut oliot muilla sivuilla, mikä mahdollistaa koko kekoa koskettavat siivoukset ilman kerääjän aiheuttamaa sivutusta. Ilman muistipaineita kerääjä toimii kuin tavallinen ikäpohjainen kerääjä, joka nuoressa sukupolvessa käyttää tavallisesti mark-and-sweep –algoritmia, vanhassa mark-compact –algoritmia, jota se käyttää myös muistin pirstoutuessa tai sivutuspaineen alla keon tiivistämiseen.

Virtuaalimuistin hallinta lähettää signaalin sivutuksen alkaessa. Sivutuslogiikka yleensä approksimoi LRU-järjestystä (*Least Recently Used*) siten se yrittää häätää ne sivut ensin, joita ei ole pisimpään aikaan käytetty. Kerääjällä on kuitenkin tarkempaa tietoa sivujen tarpeellisuudesta, siksi se voi estää joidenkin sivujen häätämistä koskettamalla niitä. Tällaisia sivuja ovat esimerkiksi nuoren sukupolven ja sisäisten tietorakenteiden sivut. Sivutuspaineen alla kerääjä yrittää ensin tyhjentää joitakin käytössä olevia sivuja tiivistämällä kekoa ja hylkäämällä (*discard*) tyhjät sivut tarpeettomina. Moni käyttöjärjestelmä (esimerkiksi Linux ja Solaris) tukee tätä toiminnallisuutta (funktion `madvise` avulla). Keinot loppuvat, jos kerääjä ei onnistu pitämään ohjelmaa muistinvaraisena, koska kerääjä ei voi vaikuttaa siihen, mitä sivuja häädetään muistista. Tämän takia kirjanmerkkikerääjää varten Hertz ja kumppanit [2005] tekivät Linux-kerneliin 2.4.20 noin kuuden sadan rivin laajennoksen, joka mahdollistaa signaloinnin ja toteuttaa funktion `vm_relinquish`. Funktion avulla kerääjä voi luovuttaa ne sivut, joita muutin tarvitsee (todennäköisesti) vähiten.

Muistipaineen alla kirjanmerkkikerääjän suoriutuminen läpäisykyvyn ja taukojen pituuden suhteen on ylivertainen muihin kerääjiin nähden. Toistaiseksi kerääjä on stop-the-world –tyyppinen, lisäksi se reagoi ainoastaan muistipaineen kasvuun mutta ei vähentymiseen, eli kekoa ei mitoiteta uudelleen vaikka fyysistä muistia olisi taas tarjolla. Siten piikki ulkoisen järjestelmän muistinkäytössä voi tiivistää pitkäksi aikaa ohjelman jalanjälkeä optimaalista pienemmäksi.

Menetelmästä saadut benchmark-tulokset antavat osviittaa siitä, että käyttöjärjestelmien virtuaalimuistinhallinta saattaa olla uuden haasteen edessä: roskienkeruu muuttaa suuresti ohjelmien muistinkäyttöä ja vie helposti pohjan nykyisiltä sivutusalgoritmeilta, mutta kehittämällä yhteistyö- ja viestintämahdollisuuksia sovellusten ja käyttöjärjestelmän välillä voidaan roskakerättyjen ohjelmien toimintaa tehostaa suuresti muistipaineen alla.

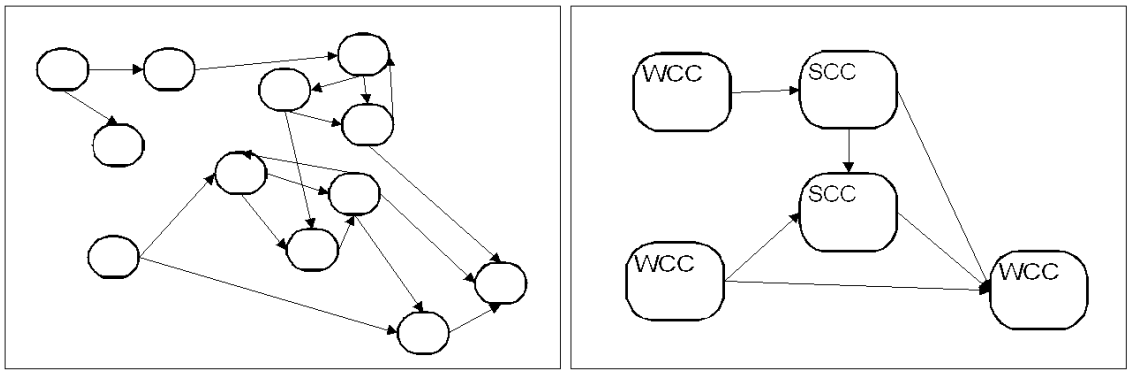
10. Viittauslaskenta CBRC-menetelmällä

Bacon ja Rajan [2001] toteavat syklisiä rakenteita purkavasta viittauslaskentaalgoritmistaan (ks. kohta 4.1), että sen tehoa voidaan kasvattaa tunnistamalla

asykliset luokat ja jättämällä niiden instanssit pois tutkittavasta joukosta. Asykliset luokat voivat sisältää skalaareja ja viittauksia asyklisiin vakioluokkiin (*final class*), sekä taulukoita, joiden alkiotyyppi on toinen edellä mainituista. Yhdistävyysanalyysin avulla voisi päästä parempiinkin tuloksiin tutkimalla ainoastaan niitä viittauksia, jotka ovat potentiaalisia syklisyyden lähteitä. Vielä paremmin saattaisi toimia menetelmä, joka eristää syklisyyden takia mahdollisesti pulmalliset oliot turvallisesti kerättävistä ja siivoaa eri joukot eri algoritmein. Tässä luvussa hahmottelen omaa CBRC-menetelmää (*Connectivity-Based Reference Counting*), joka pyrkii tähän.

10.1. CBRC-menetelmän toimintaperiaate

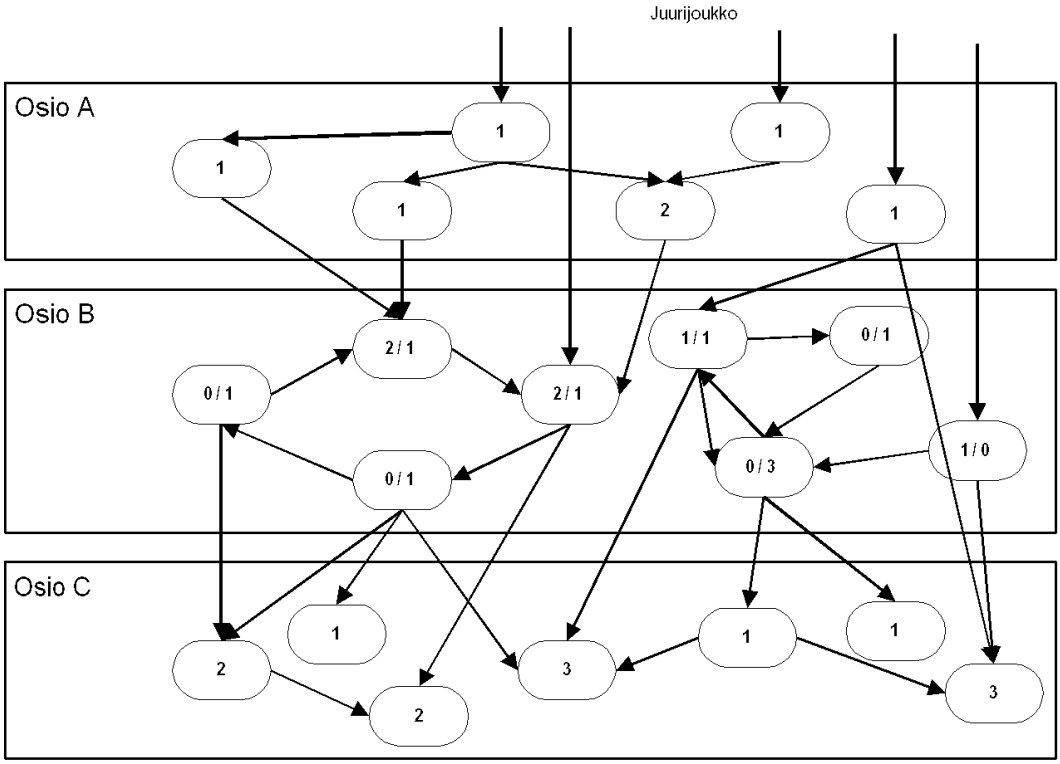
Harrisin analyysimenetelmä on omiaan erottelemaan sykliset luokkariippuvuudet asyklisistä. Keon osiointi suoritetaan CBGC-kerääjän tavoin yhdistävyyden perusteella siten, että missä tahansa analyysin tuottaman graafin osiossa on ainoastaan joko vahvasti (SCC) tai heikosti (WCC) kytkettyjä rakenteita. Näin siksi, että vahvasti kytketyt komponentit voivat muodostaa syklisiä rakenteita, mutta heikosti kytketyt eivät, ja halutaan rajoittaa mahdollisimman tiukasti ne alueet, joissa syklien etsintää suoritetaan. Osiograafin missä tahansa kohtaa voi esiintyä SCC-solmu (Kuva 7). CBRC-menetelmä ei kuitenkaan rajoitu tähän, vaan tarjoaa uuden tarkemman mekanismin, jonka avulla voidaan karsia osa roskasyklikandidaateista pois samalla kun toisia olioita voidaan pitää suuremmalla todennäköisyydellä kandidaatteina.



Kuva 7 Luokkariippuvuuskaava ja siitä Harrisin analyysillä tuotettu osiograafi.

WCC-osioissa roskat kerätään perinteisellä viittauslaskentatekniikalla, mutta SCC-osioissa viittauslaskenta tapahtuu tavallisesta hieman poiketen. Olioille ylläpidetään ylimääräistä ERC-viitelaskuria (*External Reference Count*), osion ulkopuolelta tulevia viittauksia varten. Erona nollaa suuremman tavallisen viitelaskurin ja ERC-laskurin välillä on se, että edellinen voi johtua roskasyklistä, mutta jälkimmäinen ei. Roskamääritelmä muuttuu siten, että jos

RC + ERC = 0, niin olio on roskaa. Kuva 8 havainnollistaa järjestelmää. Viittaukset WCC-osiosta SCC-osioon kirjataan tavalliseen viitelaskuriin.



Kuva 8 CBRC-osioitu keko. Osiossa B numerot tarkoittavat ERC / RC – laskureita, muissa osioissa RC-laskuria.

CBRC-algoritmi toimii samalla periaatteella kuin alkuperäinen versio muutamaa parannusta lukuun ottamatta. Kun arvoltaan yhtä suurempaa viitelaskuria vähennetään yhdellä, viittaus olioon tallennetaan taulukkoon mahdollisena roskasyklijuurena (duplikaatit ehkäistään lipun avulla). Tätä kutsun tässä luvussa "RC-1>0" -menetelmäksi. Syklejä etsittäessä kaikkia juuria tutkitaan syvyyshaulla yhtenä transitiivisena sulkeumana. Etsintöjä liipaisevat taulukon täyttyminen tai muistipaine.

CBGC-menetelmällä on seuraavia etuja perinteiseen algoritmiin nähden:

- Roskasyklejä paljastavan syvyyshaun ei tarvitse edetä osiorajojen yli, koska roskasyklejä ei voi syntyä osioiden välillä.
- Syvyyshaun ei tarvitse edetä syvemmälle saavuttaessaan oliota, jonka ERC-laskuri on nollaa suurempi.
- Osa RC-1>0 -menetelmällä saaduista roskasyklijuurista voidaan äskeiseen tapaan karsia pois tarkistamalla ERC-laskurin arvoa. Nollaa suurempi arvo ilmaisee varmuudella, että kyseessä ei ole roskasyklin elementti, sillä elementtiin viitataan juurijoukon suunnalta.

- Toisaalta nollautuva ERC-laskuri ja nollaa suurempi RC-laskurin arvo kertoo paljon todennäköisemmin roskasyklistä kuin RC-1>0 -menetelmä, sillä nimenomaan juurijoukon suunnalta tullut viittaus on poistettu.
- Äskeisen nojalla voidaan pitää melko itsestäänselvyytenä heuristista sääntöä, jonka mukaan mitä enemmän ERC-laskureita nollautuu roskia synnyttämättä, sitä enemmän osioon kertyy roskasyklejä. Tämän avulla saadaan uusi opportunistinen keruuliipaisin.
- Roskasyklien etsintä tehdään osio kerrallaan, mikä on viittaus-paikallisuuden kannalta hyvää. Lisäksi toimenpiteen aiheuttaman katkon pituutta rajoittaa osion koko.

Lyhyesti sanottuna menetelmän tehokkuus perustuu siis RC-1>0 -tapahtumien tarkempaan lajitteluun ja karsintaan paikallisuuden perusteella sekä turhien polkujen tunnistamiseen syklien etsinnässä.

10.2. Menetelmän turvallisuus

Olen esittänyt väittämiä, jotka vaikuttavat menetelmän perustana olevan syklejä havaitsevan algoritmin toimintaan. On syytä tarkistaa, että ne eivät riko kyseisen algortmin virheettömyyttä (ks. [Bacon and Rajan, 2001]). Menetelmä ei sinänsä poikkea viittauslaskennan perustekniikasta, sillä se pitää edelleen kirjaa kaikista olioihin osoittavista viittauksista muodossa RC + ERC. Siten se ei vapauta käytössä olevia olioita. Täytyy kuitenkin todistaa, että se vapauttaa kaikki ne oliot, jotka eivät ole käytössä, eli purkaa kaikki roskasyklit.

Ensimmäisen väittämän mukaan mikään olio ei voi olla osioiden välisen roskasyklin elementti, eli kaikki roskasyklit ovat osiosisäisiä. Jos olion $p \in P$ ERC-arvo on suurempi kuin 0, se johtuu viittauksesta joko juurijoukosta tai oliosta $q \in Q$, missä $Q < P$. Harris-analyysin määritelmän mukaan jos osiosta Q voidaan viitata osioon P, niin osiosta P ei voida viitata osioon Q, ei suoraan eikä välillisesti. Tämä takaa, että ei ole olemassa polkua $p \rightarrow^* q$. Toki viittaus voi tulla toisen SCC-osion roskasyklin sisältä. Menetelmä kuitenkin takaa sen, että osionsisäiset roskasyklit siivotaan, joten tällaisetkin tilanteet purkautuvat ennen pitkää.

Toisen väittämän mukaan roskasyklien etsintäjakson aikana ei tarvitse tutkia olioita, joiden ERC-laskuri on suurempi kuin nolla. Ensimmäisestä väittämästä seuraa, että nollaa suurempi ERC-laskurin arvo ei voi johtua ulkoisesta roskasyklistä. Seuraako nollaa suuremmasta ERC-laskurista, että olio ei ole muun roskasyklin jäsen? Koska RC-laskuri ei tietenkään voi saada negatiivisia arvoja ja ERC on suurempi kuin 0, roskamäärittely "olio on roskaa, jos RC + ERC = 0" takaa sen, että olio itse ei ole roskaa, ja siten siitä transitiivisesti saavutettavat oliot eivät ole roskia, eivätkä roskasyklin

elementtejä. ERC-laskurin nolla-arvo taas ei vaikuta syklien havaitsemis-algoritmiin millään tavalla, eli roskasyklit havaitaan.

10.3. Menetelmän kustannukset

Toki CBRC-menetelmällä on omat ylimääräiset kustannuksensa. SCC-osioissa ylimääräisen viitelaskurin viemä tila on ilmeinen haitta. Toisaalta vahvasti kytketyt komponentit ovat viittausjäsenmuuttujia sisältävinä yleensä melko kookkaita, joten viitelaskurin osuus olion viemästä kokonaistilasta jää suhteellisen pieneksi. Atomaarisille luokille kuten skalaareille ei tarvitse koskaan varata tilaa ERC-laskuria varten.

Myös kääntäjä joutuu tuottamaan enemmän kirjoitusmuurikoodia. Kirjoitusmuuri joutuu tarkistamaan, ylittääkö viittaus osiorajaa ja päättelemään minkä tyyppiseen osioon viitataan, jotta se voisi muuttaa oikeaa laskuria.

Muistin pirstoutuminen on kaikkien viittauslaskentamenetelmien ongelma. CBRC-menetelmä kuitenkin osioi kekoa siten, että saman luokan instanssit luodaan aina samaan osioon, mikä voi tehostaa vapaalista-varausmekanismin toimintaa ja estää muistia pirstoutumasta. Täytyy kuitenkin muistaa, että osiointi jo itsessään voi aiheuttaa jonkin verran muistin sisäistä pirstoutumista. Osiot koostuvat vakiokokoisista lohkoista, joihin voi helposti jäädä hyödyntämätöntä tilaa.

11. Roskienkeruusta tietoinen ohjelmointi

Roskienkeruusta sanotaan usein, että se vapauttaa ohjelmoijaa muistinhallinnan vastuusta kokonaan, mutta se ei sellaisenaan pidä paikkaansa. Roskienkeruun periaatteiden tuntemus on tarpeen viimeistään ohjelmointivaiheessa, mutta usein jo suunnitteluvaiheessa. Kerääjä on vastuussa tarpeettomien olioiden siivoamisesta, mutta ohjelmoijan on tehtävä selväksi, mikä on tarpeellista ja mikä ei. On tärkeää ymmärtää olioiden elinkaari: oliot syntyvät, ovat käytössä, tulevat näkymättömiksi, minkä jälkeen ovat mahdollisesti heikosti viitattuja (Java-kielessä heikosti, pehmeästi tai haamuviitattuja), viimein tulevat juurijoukosta täysin irrallisiksi, mahdollisesti viimeistellyiksi ja lopulta lakaistuiksi. Tämä pätee tutkielmassa esitettyihin kieliin (.Net, Java ja C++-kielen laajennokseen).

11.1. Muistivuodot

Muistivuodon merkitys vaihtelee ohjelmointikielestä toiseen. Esimerkiksi C ja C++ -kielissä muistivuodosta puhutaan, kun jokin olio tai olioverkon osa tulee vahingossa ohjelman kannalta näkymättömäksi (saavuttamattomaksi), kun taas roskakerätyissä kielissä asia on toisin päin: muistivuoto sattuu, kun tarpeeton olio jää ohjelman näkyviin. Kerääjä pitää huolta siitä, että näkymättömien

olioiden varaama muisti vapautetaan, mutta ohjelmoija on vastuussa siitä, ovatko elävien ja näkyvien olioiden joukot identtisiä.

Muistivuodot ovat harvinaisia roskakerätyissä kielissä, mutta niiden vaikutukset voivat olla massiivisia. Muisti voi täyttyä täysin tarpeettomista olioista. Lycklama [1999] luettelee monenlaista muistivuotolähteitä Java-ohjelmissa. Esimerkiksi Java-kielen AWT ja Swing -kirjastot ovat tunnettuja herkkinä muistivuotajina. Yleensäkin suuret ja monimutkaiset kehykset vuotavat herkästi muistia ilman, että ohjelmoija pystyy vaikuttamaan asiaan. Java-kielessä on neljä tyypillistä tilannetta, jossa oliosta voi tulla "vetkuttelija" (*loiterer*), eli turhaan näkyvänä viipyvä.

Kokoelmiin lisätyt oliot katoavat helposti ohjelmoijan silmistä, vaikkakin pysyvät kerääjän näkökulmasta näkyvissä. Kokoelmien koolle vain vapaa muisti on kattona, ja niihin on helppoa unohtaa olioita, joita joudutaan usein iteroimaan turhaan. Tällaista kokoelmaan unohdettua oliota kutsutaan nimeltä *lapsed listener*. Erityisesti staattisiin kokoelmiin kertyy paljon olioita, esimerkiksi Java 2 -alustan luokkaan java.awt.Toolkit voidaan rekisteröidä normaalisti lyhytikäsiä kuuntelijoita, jotka voivat pitää elossa laajaa olioverkkoa ohjelman suorituksen loppuun asti, ellei niitä poisteta ohjelmallisesti.

Toinen muistivuotolähde on niin sanottu viipyjäolio, eli *lingerer*. Viipyjä syntyy, kun pitkäikäisen olion viittausjäsenmuuttujasta viitataan väliaikaiseksi tarkoitettuun olioon poistamatta viittausta, kun olio on tullut tarpeettomaksi. Olio muuttuu näkymättömäksi vasta, kun toinen luodaan tilalle.

Vaikeasti tai suurin kustannuksin saatavia laskentatuloksia tallennetaan usein esimerkiksi luontimetodissa kapseloituna olion sisään varastoon. Kun olion tila muuttuu, varastoa päivitetään sitä mukaa, kun uusia laskentatuloksia tarvitaan. Tällaisia varastoituja vetkuttelevia olioita kutsutaan nimeltä *laggard*.

Viimeinen muistivuotolähde ovat pitkään kestävät metodikutsut. On tavallista luoda metodin parametriksi tarkoitettuja olioita pinomuuttujiin. Oliot annetaan parametriksi metodille, jonka suoritus voi kestää todella pitkään. Osa olioista tulee ehkä jo varhain tarpeettomiksi, mutta niitä ei voida kerätä ennen niitä luoneen pinokehyksen ponnauttamista. Etenkin pysähtelevät säikeet sisältävät pinoissaan paljon tällaisia *limbo*-olioita.

Täytyy myös muistaa, että oliot saattavat jäädä myös näkymättömyyden tilaan todella pitkiksi ajoiksi. Niin voi käydä esimerkiksi ohjelmasilmukassa luoduille olioille, jotka tulevat käytännössä kerääjän näkyviin vasta aliohjelman suorituksen päätyttyä. Tarpeettomaksi tulleet oliot on aina syytä myös merkitä eksplisiittisesti tarpeettomiksi poistamalla niihin kohdistuvat viittaukset.

11.2. Heikot viittaukset

Myös erityyppiset heikot viittaukset (*weak reference*) kertovat kerääjälle olioiden tarpeellisuudesta. Roskienkerääjä ei ota heikkoja viittauksia huomioon päättäessään olioiden tarpeellisuudesta, eli ne oliot, joihin on olemassa enää vain heikkoja viittauksia, ovat kerättävissä.

Java-kielessä viittauksen vahvuudella on neljä astetta, jotka ovat vahvuusjärjestyksessä:

- vahva, eli tavallinen viittaus (*strong reference*),
- pehmeä viittaus (*soft reference*),
- heikko viittaus (*weak reference*) ja
- haamuviittaus (*phantom reference*).

Heikkojen viittausten kaksi merkittävintä sovellusta ovat kätkömuistit (*cache*) tai heikot hakurakenteet (*weak dictionary*) ja syklisistä rakenteista johtuvien kustannusten vähentäminen [Drake, 2001]. Kätkömuistien avulla voidaan pitää kirjaa tietyistä olioista estämättä roskienkerääjää tekemästä työtään. Näin heikko viittaus voi olla ratkaisu moneen muistivuoto-ongelmaan, erityisesti *lapsed listener* -tapauksiin. Syklisten rakenteiden purku on usein kallis toimenpide. Ajatellaan esimerkiksi DOM-oliopuita, jotka ovat kahteen suuntaan linkitettyjä puurakenteita, joista oksan poistaminen on raskas iteroiva operaatio. Jos taas kaikki lehdistä juureen päin osoittavat viittaukset toteutetaan heikkoin viittauksin, haaran poistaminen tulee koostumaan yhden osoittimen nollauksesta.

Pehmeästi viitatut oliot siivotaan vain, jos vapaa muisti loppuu ja siten niiden käyttämää muistia tarvitaan muuhun. Kannattaa siis viitata pehmeästi niihin olioihin, joiden luonti ja viimeistely ovat kalliita operaatioita, mutta ovat suhteellisen harvoin käytössä. Pehmeä viittaus on omiaan ratkaisemaan *laggard*-tyyppisiä muistivuotoja.

Heikot viittaustyypit (pehmeä, heikko ja haamu) ovat Java-kielessä olioita. Heikko ja pehmeä viittaus sisältää aina kahvan viitattuun olioon, joten viitattu olio voidaan "pelastaa", eli luoda vahva viittaus siihen. Tosin olio on mahdollisesti ehtinyt kadota kerääjän toimesta, jolloin viittausoliosta saatu kahva on tyhjä.

Haamuviittaukset edustavat heikointa mahdollista viittaustyyppiä. Haamuviittauksesta saatava kahva on aina tyhjä, eli ainoastaan haamuviitattua oliota ei voi saattaa takaisin elävien joukkoon. Kerääjä käsittelee haamuviittausolioita erikoisella tavalla: jos viitattu olio on lakaistavissa ja viimeistelty, haamuviittausolio liitetään (jossain vaiheessa) jonoon, josta sovellus voi sen poimia ja käsitellä. Kerääjä ei koskaan nollaa

haamuviittausolion kahvaa, eli viitattua oliota ei lakaista ennen kuin sovellus nollaa haamuviittaukset tai haamuviittausoliot itse tulevat lakaistuiksi.

Voidaan kysyä, mitä hyötyä haamuviittauksesta on, jos viitattuun olioon ei enää pääse käsiksi? Haamuviittausmekanismi onkin enemmän sukua viimeistelylle kuin muille pehmeille viittaustavoille. Siksi haamuviittauksiin palataan seuraavassa kohdassa.

11.3. Viimeistely

Oliosuuntautuneissa kielissä luokille määritellään yleensä alustus- ja lopetusmetodit (*constructor/destructor*), esimerkiksi C++-kielessä oliot voivat lopetusmetodin avulla siivota jälkensä saman tien tullessaan elinkaarensa päähän. Roskakerätyissä kielissä olion elinkaari ei ole yhtä selkeää ja ennalta-arvattavaa, vaan vasta kun kerääjä on havainnut olion roskaksi voidaan ajoittaa oliota viimeisteltäväksi. Viimeistely suoritetaan toisessa säikeessä (eri säikeessä kuin roskienkeruu) ennaltamääräämättömään aikaan. Viimeistelijää tarvitaan todella harvoin. Sen ainoana tehtävänä on vapauttaa olion ohjelman ulkopuolelta (esimerkiksi käyttöjärjestelmästä) varaamia voimavaroja, tai muuttaa ohjelman globaalia jaettua tilaa. Viimeistelijät hidastavat muistin kierrätystä, siksi niitä ei pidä määritellä tarpeettomasti.

Boehm [2003] kiteytti lopetusmetodin ja viimeistelijän välisiä eroja: lopetusmetodin avulla voidaan taata, että pinoon varattu olio tuhotaan asiallisesti ja automaattisesti, kun sen näkyvyys päättyy, myös poikkeustilanteissa. Tässä mielessä Java-kielen lähempi vastine lopetusmetodille on `finally`-lohko, jossa `try/catch`-lohkon jäljet voidaan siivota. Viimeistelijää taas käytetään, kun kekoon varatun olion elinkaarta ei voida ennustaa ja sen loppua liittää syntaktisesti johonkin tarkkaan ohjelman suoritusvaiheeseen, silti olion varaamat resurssit on vapautettava.

Viimeistelijämetodissa olio voi "herätä kuolleista" (*resurrect*) ja herättää muita. Nimittäin on täysin laillista asettaa esimerkiksi staattinen muuttuja viittaamaan viimeisteltävään olioon, jolloin oliota ei pidetä enää roskana. Lopetusmetodi ei voi estää olion tuhoamista. Ratkaisevin ero kahden mekanismin välillä on kuitenkin se, että lopetusmetodia kutsutaan aina olion tuhoutuessa, mutta mikään ei takaa, että olion viimeistelijää suoritetaan.

Usein väitetään, että viimeistelyn asynkronisuus on roskienkeruun asynkronisuudesta johtuva haitta ja viimeistelyä pitää vältää sen virhealtiuden, huonon siirrettävyyden ja arvaamattomuuden vuoksi. Lisäksi väitetään, että lukitusta pitäisi välttää viimeistelyssä. Boehm [2003] väittää kuitenkin, että nämä väitteet ovat perusteettomia ja huomauttaa, että vaikka viimeistelyä tarvitaan harvoin, se on usein elintärkeää, kun sitä tarvitaan. Näissä tapauksissa roskienkerääjän työ olisi täysin turhaa ilman mahdollisuutta

viimeistelyyn. Vaikka viimeistelyn ajankohtaan ei ole luottamista, se ei tarkoita, ettei sovellus toimisi luotettavasti. Suurin ongelma lienee kielispesifikaatioiden puutteelliset vaatimukset viimeistelyn suhteen. Boehmin mukaan ainakin seuraavia asioita olisi syytä ottaa huomioon:

- Viimeistelijää ei suoriteta roskienkeruun tai hyötylaskennan lomassa (esimerkiksi allokaation yhteydessä).
- Viimeistelijä vaatii synkronointia, sillä sen on luultavasti tarkoitus muuttaa ohjelman globaalia jaettua tilaa.
- Monisäikeisissä ympäristöissä pitää varmistaa, että viimeistelijät suoritetaan säikeessä, joka ei omista lukkoja.
- Yhden säikeen ympäristöissä sovelluksella pitää olla valtaa saada liikkeelle viimeistelyä odottavien olioiden käsittely.
- Sovellukselle täytyy antaa mahdollisuus varmistaa, että viimeistelijää ei kutsuta liian aikaisin. Erilaisista kääntäjäoptimoinneista johtuen viimeinen osoitin olioon saattaa kadota kauan ennen kuin johonkin olion kenttään tehdään viimeinen päivitys.
- Vähäisten ulkoisten voimavarojen hallinnassa eksplisiittisesti tehtävä roskienkeruu ja viimeistely ovat hyödyllisiä, jos määritellään tarkasti vaadittavat synkronointitoimenpiteet ja otetaan huomioon viimeistelijöiden väliset mahdolliset riippuvuudet.
- Viimeistelymetodia voi suorittaa niin moneen kertaan kuin on tarpeen.

Vaikka vaatimukset tuntuvatkin itsestäänselviltä, mikään tunnettu olemassa oleva roskienkeruutoteutus ei täytä Boehmin vaatimuksia kaikilta osin. Esimerkiksi .Net ja Java -referenssitoteutuksissa olioita voidaan viimeistellä ainoastaan kerran.

Moni ympäristö tukee mekanismia, joka laittaa viimeisteltävät oliot jonoon ja antaa sovelluksen valita otollista hetkeä jonon purkuun. Java-kielessä tämä tehdään haamuviittausten avulla. Ajatellaan esimerkiksi JDBC-ajurin toteutusta. Ajurin pitää varmistaa, että varsinaiset socket-yhteydet suljetaan, aikka ohjelmoija olisikin unohtanut kutsua tietokantayhteysluokan metodia `close`. Johdetaan luokasta `java.lang.ref.PhantomReference` uusi aliluokka, joka sisältää viittauksen kyseiseen voimavaraan. JDBC-ajuri -luokkaan lisätään kaksi staattista datajäsentä, referenssijono ja lista. Jokaista uutta yhteyttä kohti luodaan uusi haamuviittausolio käyttäen uutta aliluokkaa, ja määritellään kohdejonoksi äskeinen referenssijono. Haamuviittausoliot tallennetaan myös listaan, sillä muuten ne häviäisivät seuraavassa roskienkeruussa.

Roskienkerääjä tarkistaa kaikki haamuviittaukset työnsä päätteeksi. Löytäessään roskaksi merkityn haamuviitatun olion se liittää haamuviittauksen tämän kohdejonoon. Sovellus lukee tähän tarkoitetussa säikeessä haamuviittaukset jonosta ja sulkee socket-yhteydet tarvittaessa. Viittausoliot poistetaan listasta, jotta tietokantayhteysoliot olisivat viimein lakaistavissa.

12. .Net

Microsoftin .Net -alusta on lähinnä internet-sovelluksia varten kehitetty järjestelmä. Common Language Runtime (CLR) on Microsoftin .NET-kehyksen suoritusjärjestelmä, eli se on vastuussa muistin varauksesta ja roskienkeruusta, prosessien ja säikeiden hallinnasta, turvallisuuspolitiikan toteuttamisesta ja komponenttien välisistä riippuvuuksista ohjelmaa ajettaessa. CLR suorittaa IL-kielelle (*Intermediate Language*) käännettyjä ohjelmia. Toisin kuin Java-kielen tavukoodi, IL on suunniteltu JIT-käännettäväksi. Erona Java-maailmaan on lisäksi se, että "yksi kieli monta alustaa" -asetelman sijaan .Net-kehys tukee monta kieltä (C#, C++, VisualBasic, JScript) ja yhtä alustaa (Windows).

12.1. Roskienkeruu CLR-alijärjestelmässä

Toisin kun roskienkeruu Java-kielessä, CLR-järjestelmän roskienkeruumekanismi ei ole erityisen läpinäkyvää. Microsoft ei kerro tarkkaan, miten roskienkeruu tapahtuu, mutta CLR järjestää oliot sukupolviin ja Kukreja ja Nair [2001] tietävät, että sukupolvet siivotaan mark-compact -algoritmia käyttäen. Sukupolvien lukumäärä on toteutuskohtainen vakio ja se voi myös olla yksi. Toisaalta roskienkeruuseen liittyvistä asioista vastaava luokka `System.GC` paljastaa eksoottisia tietoja olioista, esimerkiksi, sukupolven johon olion varaama muisti juuri kuuluu. Luokan avulla voidaan järjestää roskienkeruu valitussa sukupolvessa (metodilla `Collect`).

Koska CLR ei aina hallitse muistia kokonaan, vaan ulkoiset dynaamisesti ladattavat kirjastot (*DLL*) ja COM-kompponentit voivat vapaasti käyttää hallitussa muistissa olevia olioita, CLR-järjestelmää voidaan ohjeistaa jättämään joitakin olioita roskienkeruun ulkopuolelle (metodilla `KeepAlive`).

Microsoft ei myöskään paljasta, mitä kriteerejä CLR käyttää päättäessään milloin roskienkeruu suoritetaan, mutta Kukreja ja Nair [2001] tietävät edelleen, että hyötylaskenta pysähtyy kokonaan roskienkeruun ajaksi.

12.2. Viimeistely ja heikot viittaukset

Viimeistely .Net-kielessä tapahtuu seuraavalla tavalla: Kaikki viimeistelyä vaativat oliot rekisteröidään syntyessään sisäiseen tietorakenteeseen, viimeisteltävien olioiden jonoon (*Finalization queue*). Olio voidaan ohjelmallisesti poistaa jonosta (luokan `GC` metodilla `SuppressFinalize`), jos esimerkiksi

lopetustoimenpiteet tehdäänkin manuaalisesti ennen olion kuolemaa, tai lisätä olio takaisin jonoon tarvittaessa (metodilla `ReRegisterForFinalize`). Roskienkeruun merkintävaiheen aikana kerääjä tarkistaa löytyykö viimeistelyjonosta sellaisia olioita, joita se on todennut roskiksi. Kaikki tällaiset oliot siirretään varsinaiseen viimeistelyjonoon (*Freachable queue*) eikä kerääjä pidä niitä enää tässä vaiheessa roskina, vaan ne nousevat kuolleista. Tämän jälkeen muisti tiivistetään ja viimeistelyjono tyhjennetään suorittamalla jonossa olevien olioiden viimeistelijät. Viimeisteltyjen olioiden varaama tila vapautetaan vasta seuraavan roskienkeruun yhteydessä. Viimeisteltävät oliot ovat niiden luokkien instanssit, jotka ylimäärittelevät metodin `Object.Finalize`. C# ja C++ -kielissä lopetusmetodit ovat implisiittisesti viimeistelijöitä. CLR kutsuu olion viimeistelijää vain kerran.

CLR takaa, että kaikki viimeisteltävät oliot (myös elävät) viimeistellään ennen ohjelman päättymistä. Tosin joitakin poikkeustilanteita voi esiintyä, joissa viimeistelijän suoritus ei pääse loppuun tai jää kokonaan suorittamatta, esimerkiksi jos ohjelman suoritus päättyy virheen vuoksi ennenaikaisesti, tai jokin viimeistelijämetodikutsu ei palaa. CLR pyydystää viimeistelijämetodista heitetyt poikkeukset ja jatkaa seuraavan olion viimeistelyä. Lukkiutunut viimeistelyprosessi ei estä ohjelman normaalia päättymistä, vaan viimeistelyprosessi keskeytetään jos viimeisteltävien olioiden määrä ei vähene määräajassa. CLR ei takaa, että viimeistelijät suoritetaan tiettynä ajankohtana, tietyssä säikeessä tai riippuvuusjärjestyksessä.

Ympäristö tukee myös pehmeitä viittauksia luokan `WeakReference` muodossa. Viittaustyyppi on nimestään huolimatta tyypiltään nimenomaan *pehmeä* eikä *heikko*, koska heikosti viitatut oliot kerätään vasta, kun muisti on lopussa. Ohjelmoija voi päättää, siivotaanko heikosti viitattu olio vasta viimeisteltynä tai heti roskaksi toteamisen jälkeen. Luokan WeakReference oliot ovat ajoympäristön sisäisiä tietorakenteita, eikä niitä luoda varaamalla niille muistia dynaamisesti. Heikko viittaus on itse asiassa kahva sisäisen taulukon alkioon. Viimeistelytarpeensa mukaan heikot viittaukset tallennetaan yhteen kahdesta taulukosta, jotka ovat *long weak reference table* ja *short weak reference table*.

13. Java

Java on oliopohjainen, arkkitehtuurista riippumaton, verkkokäyttöön soveltuva tulkattava ohjelmointikieli ja -alusta. Roskienkeruu on pitkästä historiastaan huolimatta tullut Java-kielen mukana laajaan tietoisuuteen. Java-kielen roskienkeruuseen liittyvät yleiset piirteet ovat tulleet jo melkoisen tutuiksi

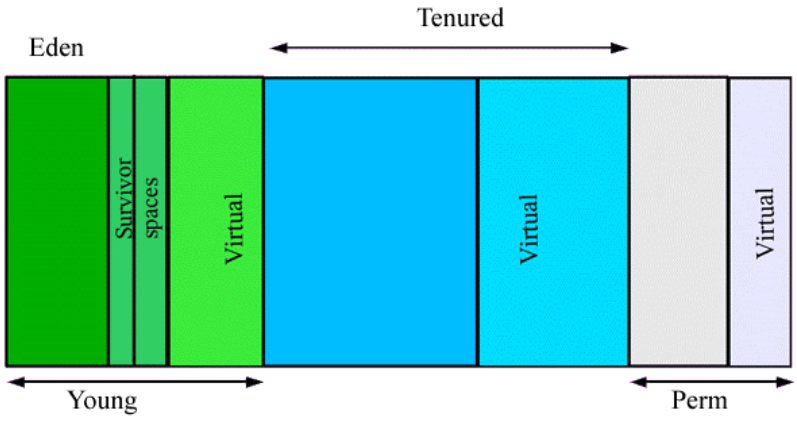
luvussa 11, siksi tässä luvussa keskitytään lähinnä virtuaalikoneen referenssitoteutuksen erityispiirteisiin.

13.1. Muistinhallinta Sun JRE-ympäristössä

Java virtuaalikoneen määrittelyn kirjoittajat Lindholm ja Yellin [1999] totetavat, että Java-kielessä on käytössä roskienkeruu, eikä olioita vapauteta koskaan eksplisiittisesti. Virtuaalikoneen toteuttaja on vapaa valitsemaan mieleisensä automaattisen muistinhallintatekniikan. Toteutukset eroavatkin toisistaan paljon. Tässä yhteydessä keskitytään Sun Microsystemsin [2002a] virtuaalikonetoteutukseen (jdk 1.4.1 - 1.5.0).

Roskienkerääjä voidaan valita Sun JRE-ympäristön Java-tulkin käynnistyksen yhteydessä neljästä eri vaihtoehdosta. Kaikki neljä kerääjää perustuvat samaan ikäpohjaiseen muistinhallintamalliin. Muisti on jaettu kahteen sukupolveen (ks. kuva 9), joiden väliin on jätetty laajenemisvaraa. Nuoren sukupolven (*young generation*) muistialue on jaettu kolmeen osaan ja sen siivoamiseen käytetään kopioivaa algoritmia. Uudet oliot allokoidaan Eedeniksi (*Eden*) nimettyyn alueeseen. Roskienkeruusta selviytyjiä varten on kahtia jaettu eloonjäänneiden osio (*survival spaces*), jossa toisessa osassa on edellisen keruun eloonjääneitä ja toinen on kopiointivara (seuraavan keruun kohdeavaruus). Oliot kopioituvat osiosta toiseen, kunnes ne ovat tarpeeksi vanhoja asettumaan aloilleen vanhan sukupolven alueella.

Kuten kuvasta 4 voidaan päätellä, suurin osa Eedenin olioista ei selviä ensimmäisestä roskienkeruusta. Vanhan sukupolven (*tenured generation*) yhteyteen on varattu erityinen pysyvien tietorakenteiden alue (*permanent space*), jossa luokka- ja metodioliot sekä muu virtuaalikoneen metadata säilytetään. Vain harvoin joudutaan koskemaan tähän osioon, mutta esimerkiksi JSP-sovellukset saattavat ladata tai generoida paljon uusia luokkia, jolloin lisätilaa tarvitaan. Vanhaa sukupolvea siivotaan mark-and-sweep -menetelmällä.



Kuva 9 Muistin osiointi Java virtuaalikoneessa [Sun, 2003].

13.2. Roskienkeruun säätäminen

Kun roskienkeruusta tulee pullonkaula, se tarkoittaa usein, että sovelluksen oliodemografia tai muistinkäyttö ei vastaa yleisimpiä oletuksia ohjelmien toiminnasta. Samaten jos halutaan korostaa tiettyä tehokkuuden aspektia, esimerkiksi jotakin kohdassa 3.1 mainittua optimoitavaa tekijää, läpäisykykyä, taukojen lyhyyttä, jalanjäljen kokoa tai muistin kierrätyksen nopeutta, löytyy paljon säädettäviä ominaisuuksia, joiden avulla oikein valitun kerääjän toimintaa voidaan tehostaa.

Oleellisin säädettävä tekijä lienee keon ja sen eri osioiden koko sekä niiden mittasuhteet. Parhaaseen läpäisykykyyn pääsee yleensä suuren nuoren sukupolven avulla, kaikkien muiden optimoitavien tekijöiden kustannuksella. Nuoren ja vanhan sukupolven oikea mittasuhde riippuu tietenkin olioiden demografiasta. Jos sovellus käyttää paljon pitkäikäisiä olioita, nuoresta sukupolvesta ei kannata tehdä kovin isoa, muuten vanhalle sukupolvelle jää liian vähän tilaa.

Katkojen pituuden vähentämisessä auttaa pieni nuori sukupolvi, jota siivotaan lyhyessä ajassa, mutta varsin tiheään. Jalanjälkeä voidaan säätää asettamalla keon koolle ala- ja yläraja sekä prosentuaalinen minimi- ja maksimiarvo vapaalle muistille, jolloin virtuaalikone kasvattaa tai kutistaa kekoa tarpeen mukaan.

Nuoressa sukupolvessa Eedenin ja eloonjääneiden osioiden kokosuhde on oletusarvoisesti 2:6, eli roskienkeruussa kohdeavaruus on lähdeavaruuden kahdeksasosa. Jos kohdeavaruus osoittautuu liian pieneksi, eloonjääneet ylivuotavat suoraan vanhaan sukupolveen. Ylivuoto voi eskaloitua helposti koko kekoa koskevaksi roskienkeruuksi. Huonosti mitoitettu nuori sukupolvi voi helposti johtaa tilanteeseen, jossa koko kekoa joudutaan siivoamaan joka kerta. Näin käy helposti, kun nuoren sukupolven koko lähenee 50% keon koosta. Kerääjä ylentää eloonjääneet vanhaan sukupolveen vasta kun niiden ikä on saavuttanut kynnysarvon. Kynnysarvoa säädetään automaattisesti sellaiseksi, että eloonjääneiden osiot olisivat puolityhjiä.

Säätämisen tekevät mahdolliseksi JRE-ympäristön antamat tarkat diagnostiikkatiedot roskienkeruusta. Komentoriviparametrien avulla voidaan saada listaus jokaisen roskienkeruun ajankohdasta ja kestosta, kohdesukupolvesta ja sen täyttöasteesta, lakaistujen roskien määrästä, ylenemisen kynnysiästä, yms.

13.3. Roskienkerääjät

Jos oletuskerääjä ei ole riittävän tehokas sovelluksen tarpeisiin, voidaan ottaa optimoitu kerääjä tilalle. Läpäisykykyä voidaan monisuoritinlaitteissa lisätä *throughput collector* –kerääjän avulla, joka tukee myös adaptiivisia säätöjä:

kerääjä voi hyödyntää toiminnassaan pitkältä ajalta automaattisesti keräämänsä tilastollista aineistoa sovelluksen käyttäytymisestä. Ainoastaan nuoren sukupolven siivous suoritetaan monen säikeen voimin. Säikeitä on oletusarvoisesti yhtä monta kuin suorittimia. Suorittimia pitää käytännössä olla enemmän kuin kaksi, jotta kaikki hyöty saataisiin tästä kerääjästä irti. Pääsyä vanhaan sukupolveen ei ole synkronoitu, siksi eri säikeet ylentävät omiin puskureihinsa vanhan sukupolven alueella, mikä voi johtaa muistin pirstoutumiseen.

Katkojen pituutta voidaan lyhentää samanaikaisen kerääjän (*concurrent collector*) avulla. Kerääjä ajetaan omassa säikeessään ja se keskittyy siivoamaan vanhaa sukupolvea. Ohjelman muut säikeet pysähtyvät keruun aikana kahdesti ja silloinkin vain lyhyeksi ajaksi. Jos ohjelman muut säikeet pysähtyvät kokonaan roskienkeruun ajaksi, säätöjä täytyy muuttaa.

Toinen vaihtoehto katkojen lyhentämiseksi on vähittäinen kerääjä (*incremental* tai *train collector*). Tämä kerääjä sopii koneisiin, joissa on vain yksi suoritin. Vanhan sukupolven siivouksen vaatima aika jaetaan pienempiin osiin siivoamalla vanhaa sukupolvea vähittäin nuoren sukupolven siivouskertojen yhteydessä. Menetelmä vaatii niin paljon koordinointia, että hyötylaskennan osuus suoritusajasta vähenee väistämättä. Jos tästä ei ole suurempaa haittaa, vähittäisen kerääjän käyttö on tehokas tapa parantaa ohjelman vasteaikoja.

14. C++

Perinteisten imperatiivisten kielten tavoin C ja sen olioperustainen laajennos C++ käyttävät eksplisiittistä muistinhallintaa. Molemmat kielet ovat varsin hankalia roskienkeruun kannalta, sillä ne eivät ole tyyppitietoisia (*type accurate*), eli muuttujatyyppejä ei tunneta ajoaikana.[4] Kerääjän täytyy kuitenkin pystyä erottelemaan atomaarinen data osoittimista, jotta olioverkon läpikäynti (tai viittauslaskenta) olisi mahdollinen. C ja C++-kielten yhteydessä voidaan siis käyttää oletusarvoisesti ainoastaan konservatiivisia kerääjiä, jotka pitävät osoittimena kaikkia tietorakenteita, jotka näyttävät osoittimelta.

14.1. Libgc

Tyypillisesti konservatiivinen kerääjä korvaa suoritusjärjestelmän muistinvarausfunktiot (*malloc*, yms.) omalla toteutuksellaan ja suorittaa keruun allokoinnin yhteydessä. Muistin vapautusoperaatio (*free*) korvataan yleensä tyhjällä funktiolla.

[4] Tosin C++-kieleen on olemassa heikosti tuettu RTTI-laajennos (*Run-time Type Information*), joka mahdollistaa objektin tietotyypin tarkistamisen ajon aikana.

Tunnetuin olemassaoleva C/C++ -kerääjä lienee Boehm-Demers-Weiser -kerääjä (kyseessä [Boehm *et al.*, 2003]), joka tunnetaan myös libgc-kirjastona. (Sun Microsystemsillä on oma libgc-kirjastonsa.) Libgc-rajapinnan kautta voidaan muiden muassa rekisteröidä oliolle viimeistelijä, varata muistia, joka voi olla atomaarista (josta ei etsitä osoittimia) tai pysyvää (jota ei lakaista). Muistia ei tarvitse vapauttaa, koska sen hoitaa mark-and-sweep -kerääjä. Konservatiivuudesta johtuvan epävarmuuden takia olioita ei liikuteta koskaan muistissa. Libgc tukee tunnetuimmat alustat (Unix, Linux, Windows, MacOS X).

14.2. Roskienkeruu osaksi C++-kieltä?

Viimeisen vuosikymmenen aikana on esitetty useita kokonaisvaltaisia ratkaisuja ja ehdotuksia roskienkeruun integroimiseksi C++-kieleen. Ellis ja Detlefs [1993] esittivät C++-kieleen roskienkeruujärjestelmän, joka lienee tunnetuin ja kattavin ratkaisu, josta on myös otettu paljon mallia (yhtenä esimerkkinä libgc-kirjasto). Esitys pohjautuu Bartlettin (1989) ja Boehmin (1991) tekemään esityöhön ja Zornin tehokkuusmittauksiin (1992).

Ellis ja Detlefs lähestyivät asiaa holistisesti järjestelmätasolla ja määrittelivät järjestelmälle neljä osa-aluetta, jotka ovat

- rajapinta ohjelmointikieleen (*language interface*),
- turvallisten ohjelmarakenteiden alijoukko (*safe subset*),
- käytettävät algoritmit (*collection algorithms*) ja
- turvallisen koodin generointi (*code generation safety*).

Esityksen kulmakivinä ovat järjestelmälle asetetut turvallisuusvaatimukset, vaatimukset mahdollisimman pienistä muutoksista kieleen, sen toteutuksiin ja ohjelmointityyliin sekä perinteistä muistinhallintaa käyttävien ja roskienkeruun alaisten moduulien yhteensopivuus. Myös toteutuksen kannettavuuteen ja tehokkuuteen kiinnitetään huomiota.

Turvallisen koodin generointi taataan kahden käännösaikaisen rajoitteen, kymmenen käännösaikaisen ja kuuden ajoaikaisen tarkistuksen avulla. Kääntäjien ja ajoaikaisten järjestelmien täytyy toteuttaa nämä rajoitteet ja tarkistukset, jotta ne olisivat turvallisia roskienkeruun kannalta (*GC-safe*). Kaikki tarvittavat toimenpiteet voidaan tehdä välikoodin generoinnin yhteydessä (*front-end compilation*).

Turvallisten ohjelmarakenteiden alijoukko on C++-kielen syntaksin alijoukko, jota käyttämällä roskienkeruun virheettömyys voidaan taata. Ohjelmoija merkitsee erityisellä `pragma`-määreellä ne lähdekoodin osat, jotka on tarkoitettu roskienkeruuturvallisiksi.

```
#pragma safe
```

```
//turvallinen koodi

#pragma unsafe
```

Kääntäjä huolehtii siitä, että kaikki ohjelmarakenteet ovat turvallisia ja turvallisuussäännöt pysyvät voimassa myös ajoaikana. Tämä tarkoittaa käytännössä suuren joukon ylimääräisiä ajoaikaisia tarkistuksia, esimerkiksi osoitinmuuttujien alkuarvon ja arvomuutosten validiteetin osalta.

Roskienkeruun alaiset oliot erotetaan muista olioista (*collected / non-collected objects*) omaan (loogiseen) kekoonsa kieleen lisätyn `gc`-määreen avulla. Määrettä käytetään luokan tai muuttujan esittelyn yhteydessä siten, että osoitin `T*` on identtinen osoittimen `gc T*`:n kanssa. Toisin sanoen määre ei vaikuta tyypitykseen tai tyypinmuunnoksiin, mikä takaa yhteensopivuuden keräämättömien moduulien kanssa.

Muista moduuleista tuoduista luokista voi luoda kerättäviä instansseja `new`-operaattorilla, tai yleisesti `typedef`-operaattorin avulla:

```
new gc T;

typedef gc T TGC;

new TGC;
```

Seuraava asetelma on myös mahdollinen:

```
gc class D : C {...};
```

Luokka C on tuotu moduulista, jossa ei ole valmiutta käyttää roskienkeruuta tai toimia monisäikeisessä ympäristössä. Vaarana on, että luokan C lopetusoperaatio suoritetaan asynkronisesti roskienkeruun yhteydessä tuntemattomin seurauksin. Tällöin voidaan määritellä luokalle D jono, johon viimeisteltävät oliot kerätään. Ohjelma voi tyhjentää jonon turvallisessa suorituksen vaiheessa.

Kerättäviä olioita voi vapauttaa `delete`-operaattorilla, mutta varsinaisen viimeistelytyön tekee kerääjä. Lopetusoperaattori toimii oletusarvoisena viimeistelijänä, mutta viimeistelijämetodi voidaan määritellä kerättäville luokille erikseen. Roskienkeruuta tukevat luokat toteuttavat automaattisesti yleisen rajapinnan, joka sisältää muun muassa kaikki yllämainitut viimeistelyyn liittyvät operaatiot. Myös heikot viittaukset toteutetaan vastaavalla tavalla.

Roskienkeruun toimivuus Javassa on antanut puhtia pyrkimyksille lisätä roskienkeruu C++-kieleen ja joitakin hyviä toteutuksia on jo nyt saatavilla. Valitettavasti ANSI-standardointia joudumme luultavasti odottamaan vielä todella pitkään.

15. Yhteenveto

Automaattisen muistinhallinnan käyttö ohjelmistotuotannossa on yleistymässä. Koska roskienkeruu on usein järjestelmän toimivuuden pullonkaulana, algoritmin valinnalla ja hienosäädöllä on ratkaiseva vaikutus siihen, pystyykö ohjelma täyttämään sille asetetut vaatimukset. Vaatimuksia voidaan asettaa läpäisykyvyn, jalanjäljen, muistin saatavuuden ja vasteaikojen suhteen. Vasta kun näiden tekijöiden tärkeyssuhde on määritelty, voidaan etsiä tilanteeseen sopiva algoritmi. Tärkeää on myös roskienkeruumekanismien huomioon ottaminen ohjelman suunnittelu- ja toteutusvaiheessa.

Algoritmien esittelyn yhteydessä esiteltiin myös niiden käyttöön liittyviä ongelmia. Esimerkiksi merkitse ja lakaise –tyyppisten algoritmien ongelmana on muistin pirstoutuminen ja olioiden hajaantuminen muistissa. Ongelmaan tuovat helpotusta algoritmin tiivistävät muunnelmat, jotka vaativat kuitenkin enemmän laskentaa. Laskentakustannuksia voidaan vähentää käyttämällä kopioivia menetelmiä, jolloin maksettava hinta on pahimmillaan kaksinkertaistuva muistintarve. Viittauslaskentatekniikan kompastuskivenä ovat sykliset tietorakenteet, mutta niiden purku ei ole ratkaisematon ongelma.

Monessa sovelluksessa vasteajat ovat kriittinen asia, esimerkiksi hyvän käyttäjäkokemuksen tai reaaliaikaisuusvaatimusten vuoksi. Vähittäisen roskienkeruumenetelmän valinnassa huomioitavat seikat ovat

- menetelmän sallimat vasteajat,
- algoritmin huolellisuus, eli roskienkeruun päättyessä jäljelle jääneen kelluvan roskan määrä,
- tarpeettomaksi muuttuneen muistin vapauttamiseen kuluva aika ja
- vähittäisyyden tuoma ylimääräinen laskentakuorma.

Viittauslaskentatekniikkaa käyttämällä voidaan saavuttaa melko suurta vähittäisyyden astetta. Muiden kanonisten menetelmien tapauksessa vähittäisyyteen päästään lähinnä osioimalla muistia erillisiksi roskakerättäviksi yksiköiksi. Suosituin osiointiperuste on olioiden ikä.

Ikäpohjaisten menetelmien käyttö on yleensä kannattavaa, koska olettamus olioiden lyhytikäisyydestä pätee lähes aina. Menetelmien tehokkuuteen voivat vaikuttaa oleellisesti säädettävät ominaisuudet ja optimoitavat tekijät, muiden muassa

- sukupolvien lukumäärä ja
- yksittäisten sukupolvien osuus käytettävissä olevasta muistista,
- kynnysiät,
- eri sukupolvissa käytetyt algoritmit,
- sukupolvien rajoja ylittävien viittausten rekisteröinti,
- roskienkeruun eskaloituminen sukupolvesta toiseen sekä

- menetelmän tai menetelmien vähittäisyys.

Koska oliot kuolevat nuorena, vanhojen olioiden joukossa on vähän roskia. Siksi niitä ei kannata kopioida, mutta nuorten olioiden kopiointi taas parantaa viittauspaikallisuutta sijoittamalla kytköksissä olevat oliot lähekkäin. Olioita ei yleensä luoda turhaan, vaan vastaluodut oliot ovat usein ohjelman kannalta (hetkellisesti) kaikkein tarpeellisimmat. Tästä syystä nuori sukupolvi sisältää sekä paljon roskia että hyvinkin tarpeellisia olioita. Siivoukset on siis syytä järjestää niin, että nuorilla olioilla on tarpeeksi aikaa kuolla. Osioimalla kekoa oikein, esimerkiksi pitämällä nuorten olioiden joukkoa pienenä, päästään ikäpohjaisten menetelmien avulla yleensä suureen vähittäisyyden asteeseen.

Tehokkaimmat menetelmät etsivät roskia opportunistisesti, pyrkimällä ennustamaan, mistä ja milloin eniten löytyy roskia. Ikäpohjaisen roskienkeruun ongelmana on sovelmien olioikäjakauman vaihtelevuudesta johtuva ennustamattomuus. Olioiden väliset kytkökset näyttävät antavan parempia vinkkejä siitä milloin olioita kuolee paljon, sillä mitä vahvemmin kaksi oliota ovat kytköksissä toisiinsa, sitä todennäköisemmin ne kuolevat yhdessä. Avainasemassa olevat oliot kuollessaan vievät paljon olioita mukanaan. Tämä heuristiikka on yhdistävyyspohjaisen roskienkeruun opportunismin taustalla.

Tutkielmassa hahmoteltiin myös CBRC-menetelmää, eli uutta viittauslaskenta-algoritmia, joka hyödyntää yhdistävyysanalyysiä roskasyklien eliminoimisessa. Lisätutkimuksen arvoinen asia olisi CBRC-menetelmän toteutus Jikes RVM -ympäristössä, jotta sen tehoa voisi verrata vastaaviin menetelmiin SPEC-benchmark -testisarjan avulla.

Automaattinen muistinhallinta helpottaa ohjelmoijan työtä, mutta asettaa samalla uusia haasteita. Manuaalista muistinhallintaa käytettäessä ohjelmoija joutuu kiinnittämään erityistä huomiota tarpeettomien olioiden vapauttamiseen. Yleisesti ajatellaan, että tähän ei ole tarvetta kiinnittää huomiota automaattisen muistinhallinnan ollessa käytössä, mutta se ei pidä paikkaansa. Ohjelmoijan täytyy huolehtia edelleen siitä, että kaikki viittaukset poistetaan tarpeettomiin olioihin, jotta roskienkerääjä voisi tehdä työtään. Tehokkuuden nimissä ohjelmoijan on usein syytä olla vuorovaikutussuhteessa roskienkerääjän kanssa, esimerkiksi erilaisten heikkojen viittausten ja viimeistelijöiden kautta.

Automaattisen muistinhallinnan perusolettamukset poikkeavat suuresti eri ympäristöissä. Java-kielessä roskienkeruu on läsnä varsin abstraktilla tavalla, ainoastaan viimeistelyn ja heikkojen viittaustyyppien kautta. Kielen spesifikaatiossa ei oteta kantaa roskienkeruun toteutukseen, esimerkiksi käytettävään algoritmiin, vaan se on toteuttajan valittavissa. Referenssitoteutus tarjoaa virtuaalikoneen käynnistyksen yhteydessä mahdollisuuden valita

roskienkeruumenetelmä ja tehdä siihen monenlaisia säätöjä. Sen sijaan .Net-alustalla on ikäpohjainen roskienkeruu kiinteästi käytössä, mikä heijastuu suoritusjärjestelmän rajapintakirjastoihin ja siten jopa lähdekoodiin.

Vaikka suurin osa ohjelmistokehityksestä tehdäänkin nykyään Java ja .Net -alustoilla, ohjelmia ja komponentteja tuotetaan edelleen perinteisellä tavalla, lähinnä C ja C++-kielellä. Roskienkeruuta voidaan soveltaa myös näihin kieliin ajoaikaisen kirjaston avulla. C++-kielen standardoinnista vastaavalle komitealle on esitetty laajennusehdotus kieleen, joka tekisi turvallisen ja tehokkaan roskienkeruun mahdolliseksi myös C++-ohjelmissa.

Viiteluettelo

[Alpern *et al.*, 2000] B. Alpern, O, D. Attanas, J. Barton, M. Burke, P. Cheng, J. Choi, A. Cocchi, S. Fink, D. Grove, M. Hind, S. Hummel, D. Lieber, V. Litvinov, T. Ngo, I. Mergen, V. Sarkar, M. Serrano., J. Shepherd, S. Smith, V. Sreedhar, H. Srinivasan, and J. Whaley, The Jalapeño virtual machine. *IBM Systems Journal*, **39** (1), February 2000.

[Appel, 1989] A. W. Appel, Simple generational garbage collection and fast allocation. *Software Practice and Experience*, **19** (2), 1989, 171-183.

[Bacon and Rajan, 2001] D. F. Bacon and V. T. Rajan, Concurrent cycle collection in reference counted systems, In J. L. Knudsen (editor), *Proceedings of 15th European Conference on Object-Oriented Programming, ECOOP 2001,* Budapest, Hungary, June 18-22, volume 2072 of *Lecture Notes in Computer Science*, Springer-Verlag, 2001, 207–235.

[Bacon *et al.*, 2003] D. F. Bacon, P. Cheng, and V. T. Rajan, A real-time garbage collector with low overhead and consistent utilization. *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Press, 2003, 285-298.

[Baker, 1978] H. G. Baker, List processing in real time on a serial computer, *Communications of the ACM*, **21** (4), Apr. 1978, 280-294.

[Baker, 1992] H. G. Baker, The Treadmill, real-time garbage collection without motion sickness, *ACM SIGPLAN Notices*, **27** (3), March 1992, 66-70.

[Blackburn *et al.*, 2002] S. M. Blackburn, P. Cheng, and K. S. McKinley, Beltway: getting around garbage collection gridlock. *ACM SIGPLAN Notices, Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, **37** (5), May 2002.

[Blackburn *et al.*, 2004] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. *ICSE 2004, 26th International Conference on Software Engineering*, Edinburgh, May 2004.

[Blackburn and McKinley, 2003] S. M. Blackburn and K. S. McKinley, Ulterior reference counting: fast garbage collection without a long wait, *ACM SIGPLAN Notices, Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, **38** (11), October 2003.

[Boehm, 2003] H-J. Boehm, Destructors, finalizers, and synchronization. *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Press, 2003, 262-272.

[Boehm *et al.*, 2003] H. J. Boehm, A. Demers, M. Weiser. A garbage collector for C and C++. Available at http://www.hpl.hp.com/personal/Hans_Boehm/gc/index.html [31.10.2003].

[Cannarozzi *et al.*, 2000] D. Cannarozzi, M. Plezbert, and R. Cytron, Contaminated garbage collection. Programming Languages Design and Implementation (PLDI), 2000.

[Cheney, 1970] C. J. Cheney, A nonrecursive list compacting algorithm, *Communications of the ACM*, **13** (11), Nov. 1970, 677-678.

[Choi *et al.*, 1999] J.-D. Choi, M. Gupta, M. Serrano, V. Sreedhar and S. Midkiff, IBM T.J. Watson Research, Escape Analysis for Java. *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1999, 1-19.

[Christopher, 1984] T. W. Christopher, Reference count garbage collection. *Software – Practice and Experience*, **14** (6), June 1984, 503–507.

[Cohen, 1981] J. Cohen, Garbage collection of linked data structures. *Computing Surveys*, **13** (3), September 1981, 341-367.

[Deutsch and Bobrow, 1976] L. P. Deutsch, D. G. Bobrow, An efficient incremental automatic garbage collector, *Communications of the ACM*, **19** (9), September 1976, 522-526.

[Dieckmann and Hölzle, 1999] S. Dieckmann and U. Hölzle, A study of allocation behavior of the SPECjvm98 Java benchmarks. *European Conference for Object-Oriented Programming (ECOOP)*, 1999.

[Dijkstra *et al.*, 1978] E. W. Dijkstra, L. Lamport, A. J. Martin and E. F. M. Steffens, On-the-fly garbage collection: An exercise in cooperation, *Communications of the ACM*, **21** (11), 966-975.

[Drake, 2001] F. L. Drake, Jr., Weak References (Python Enhancement Proposal). Available at http://www.python.org/peps/pep-0205.txt [11.1.2001].

[Eckel, 2000] B. Eckel, Thinking in Java. Available at http://www.codeguru.com/java/tij/tij_c.shtml [15.12.2003].

[Ellis and Detlefs, 1993] J. R. Ellis and D. L. Detlefs, Safe, Efficient Garbage Collection for C++. Available at http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-102.html [10.6.1993].

[Harris, 1999] T. Harris, Early storage reclamation in a tracing garbage collector, *ACM SIGPLAN Notices*, April 1999.

[Hayes, 1991] B. Hayes, Using key object opportunism to collect old objects. Object-Oriented Programming, Systems, Languages, and Applications, 1991.

[Hertz *et al.*, 2005] M. Hertz, Y. Feng, E. D. Berger, Garbage Collection Without Paging, *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, **40** (6), 2005.

[Hirzel *et al.*, 2002] M. Hirzel, J. Henkel, A. Diwan, and M. Hind, Understanding the connectivity of heap objects. International Symposium on Memory Management (ISMM), 2002.

[Hirzel *et al.*, 2003] M. Hirzel, A. Diwan, M. Hertz, Connectivity-Based Garbage Collection, *Proceedings of the 18th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications,* **38** (11), October 2003.

[Hudson and Moss, 1992] R. L. Hudson and J. E. B. Moss, Incremental garbage collection for mature objects. In: Yves Bekkers and Jacques Cohen (eds.), *Proceedings of the First International Workshop on Memory Management, IWMM'92*, volume 637 of *Lecture Notes in Computer Science*, St. Malo, France, September 1992, Springer-Verlag.

[Johnstone and Wilson, 1998] The memory fragmentation problem: solved?, *ISMM'98 Proceedings of the ACM SIGPLAN International Symposium on Memory Management*, 1998, 26-36.

[Kukreja and Nair, 2001] A. Kukreja and R. Nair, Garbage Collection in .Net. Available at http://www.dotnetextreme.com/articles/garbageCollector.asp [31.10.2003].

[Levanoni and Petrank, 2001] Y. Levanoni and E. Petrank, An on-the-fly reference counting garbage collector for Java. *ACM Conference Proceedings on Object–Oriented Programming Systems, Languages, and Applications*, Tampa, FL, October 2001, 367–380.

[Lindholm and Yellin, 1999] Tim Lindholm and Frank Yellin, *The Java Virtual Machine Specification, Second Edition*, Addison-Wesley, 1999.

[Lins, 1992] R. D. Lins, Cyclic reference counting with lazy mark-scan, *Inf. Process. Lett*, **44** (4), December 1992, 215–220.

[Lycklama, 1999] E. Lycklama, Does Java Technology Have Memory Leaks? Available at: http://www.cs.kent.ac.uk/people/staff/rej/lycklama99.pdf [31.10.2003].

[Martínez *et al.*, 1990] A. D. Martínez, R. WachenChauzer, and R. D. Lins, Cyclic reference counting with local mark-scan, *Inf. Process. Lett.*, **34** (1), 1990, 31-35.

[Moon, 1984] D. A. Moon, Garbage collection in a large LISP system, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, ACM Press, August 1984, 235-245.

[Park and Goldberg, 1992] Y. G. Park and B. Goldber, Escape analysis on lists. Programming Languages Design and Implementation (PLDI), 1992.

[Qian and Hendren, 2002] F. Qian and L. Hendren, An adaptive, region-based allocator for Java. International Symposium on Memory Management (ISMM), 2002.

[Reinhold, 1994] M. B. Reinhold, Cache Performance of Garbage-Collected Programs, *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1994, 206-217.

[Soman *et al.*, 2003] S. Soman, Ch. Krintz, and D. F. Bacon, Adaptive, application-specific garbage collection, University of California, Santa Barbara, Tech. Rep., March 2003.

[SPEC, 1999] Standard Performance Evaluation Corporation. SPECjvm98 Documentation, release 1.03 ed., March 1999.

[SPEC, 2001] Standard Performance Evaluation Corporation. SPECjbb2000 (Java Business Benchmark) Documentation, release 1.01 ed., 2001.

[Stefanović *et al.*, 1999] D. Stefanović, K. S. McKinley and J. E. B. Moss, Age based garbage collection. *Proceedings of SIGPLAN 1999 Conference on Object Oriented Programming Languages & Applications*, **34** (10), ACM Press, 1999, 379-381.

[Stefanović *et al.*, 2000] D. Stefanović, K. S. McKinley, J. E. B. Moss, On models for object lifetime distributions. International Symposium on Memory Management (ISMM), 2000.

[Stefanović *et al.*, 2002] D. Stefanović, M Hertz, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Older-first Garbage Collection in Practice: Evaluation in a Java Virtual Machine, *ACM SIGPLAN Notices, Proceedings of the 2002 workshop on Memory system performance*, **38** (2) supplement, June 2002.

[Sun, 2003] Sun Microsystems Inc, Tuning Garbage Collection with the 1.4.2 Java™ Virtual Machine, (performance documentation for the java hotspot virtual machine). Available at http://java.sun.com/docs/hotspot/gc1.4.2/index.html, [18.09.2003].

[Sun, 2002a] Sun Microsystems Inc, Java 1.4.1 HotSpot Virtual Machine White Paper, (Technical White Paper september 2002). Available at http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/Java_HSpot_WP_v1.4.1_1002_1.html [12.4.2003].

[Sun, 2005] Sun Microsystems Inc, J2SE 1.5.0 Documentation, Available at http://java.sun.com/j2se/1.5.0/docs/guide/vm/gc-ergonomics.html

[Ungar, 1984] D. Ungar, Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *Practical Software Development Environments*, 1984.

[Wilson, 1992] P. R. Wilson. Uniprocessor garbage collection techniques. *Proceedings of the First International Workshop on Memory Management, IWMM'92*, volume 637 of *Lecture Notes in Computer Science*, St. Malo, France, September 1992, Springer-Verlag.

[Wilson *et al.*, 1992] P. R. Wilson, M. S. Lam and T. G. Moher, Caching considerations for generational garbage collection. *Conference on Lisp and Functional Programming*, ACM, 1992, 32-42.

[Zorn, 1991] B. G. Zorn, The Effect of Garbage Collection on Cache Performance, Technical report **CU-CS-528-91**, Department of Computer Science, University of Colorado at Boulder, May 1991.